POLITECNICO
MILANO 1863

Design principles

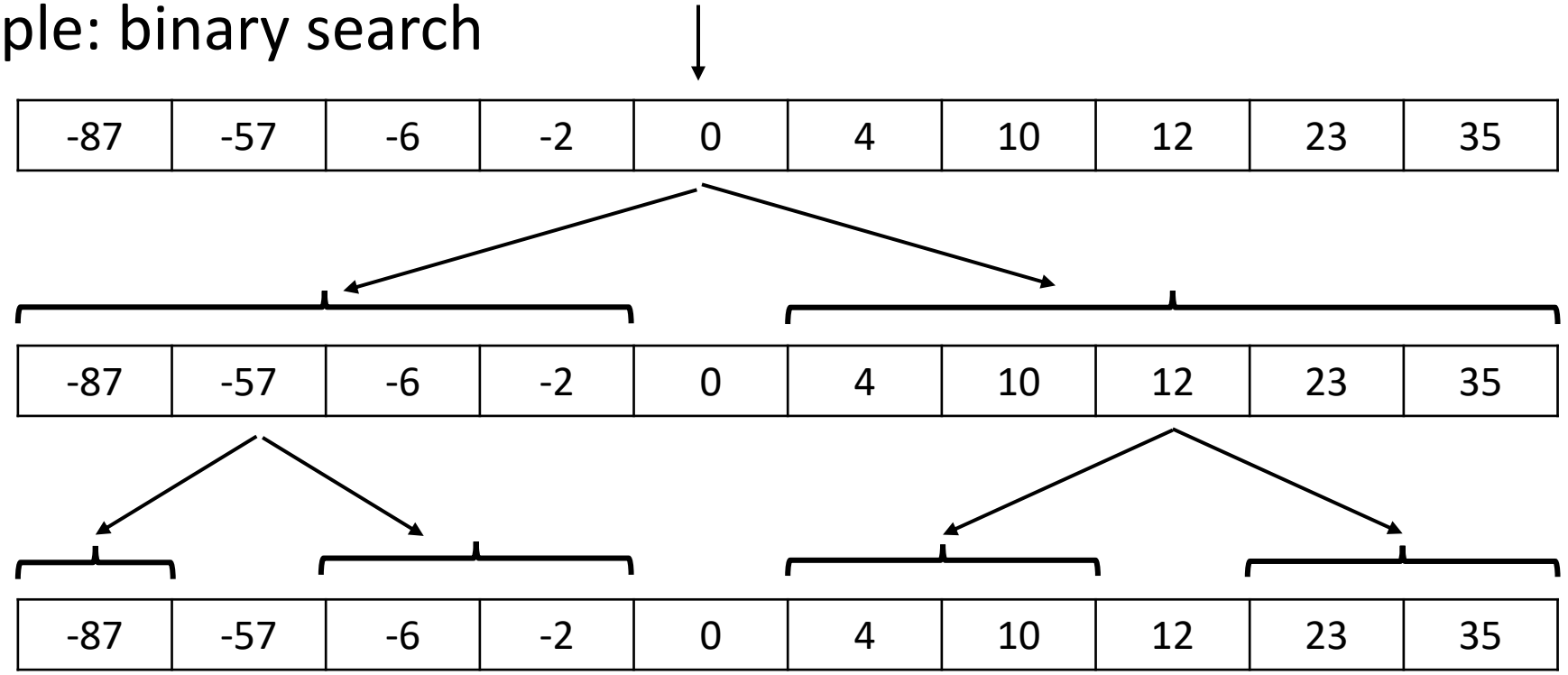
Design Principles

POLITECNICO
MILANO 1863

- Design Principle 1: Divide and conquer
- Design Principle 2: Keep the level of abstraction as high as possible
- Design Principle 3: Increase cohesion where possible
- Design Principle 4: Reduce coupling where possible
- Design Principle 5: Design for reusability
- Design Principle 6: Reuse existing designs and code
- Design Principle 7: Design for flexibility
- Design Principle 8: Anticipate obsolescence
- Design Principle 9: Design for portability
- Design Principle 10: Design for testability
- Design Principle 11: Design defensively

Design Principle 1: Divide & Conquer

- We use recursive decomposition to divide a problem in simpler subproblems
- Example: binary search



Design Principle 2: Keep the level of abstraction as high as possible

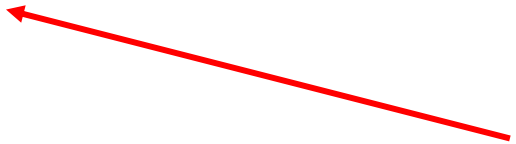
POLITECNICO MILANO 1863

- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity

  - A good abstraction is said to provide *information hiding*
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

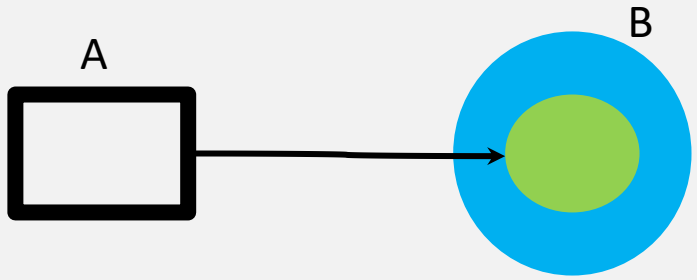
Design Principle 3: Cohesion

- Example of a non-cohesive module
  - Class Utility {
    - ComputeAverageScore(Student s[])
    - ReduceImage(Image i)
  - }

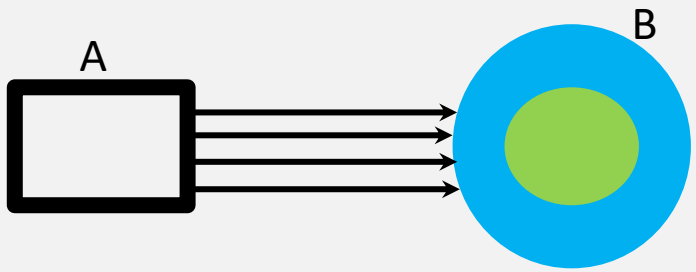


Is this a cohesive module???

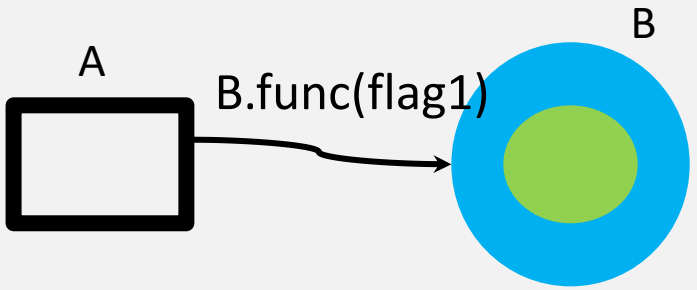
Design Principle 4: Coupling
What to avoid



A is accessing the data structure in B breaking encapsulation
→ *content coupling*



Do we really need so many messages from A to B? -> *communication coupling*




```
class b {
  func(flag f) {
    if(f == flag1) do this
    else if (f == flag2) do that
    else…
  }
} → control coupling
```

Coupling and interaction overhead

POLITECNICO MILANO 1863

- Interaction overhead is often due to
  - High interaction frequency
  - High volume of data exchanged during interaction
  - Resource contention
  - Interleaved communication and computation
- In all these situations high coupling plays an important role

Design Principle 5: Design for reusability

POLITECNICO MILANO 1863

- Design the various aspects of your system so that they can be used again in other contexts
  - Generalize your design as much as possible
  - Simplify your design as much as possible
  - Follow the preceding all other design principles
  - Design your system to be extensible

Design Principle 6: Reuse existing designs and code

- Design with reuse is complementary to design for reusability
  - Take advantage of the investment you or others have made in reusable components
  - NOTE: Cloning should not be seen as a form of reuse

Design Principle 7: Design for flexibility

POLITECNICO MILANO 1863

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Use reusable code and make code reusable
  - Do not hard-code anything

Design Principle 8: Anticipate obsolescence

POLITECNICO MILANO 1863

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Do not rush using early releases of technology
  - If possible
    - Avoid using software libraries that are specific to particular environments
    - Avoid using undocumented features or little-used features of software libraries
    - Avoid using software or special hardware from companies that are less likely to provide long-term support
    - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

POLITECNICO MILANO 1863

- Have the software run on as many platforms as possible
  - Avoid, if possible, the use of facilities that are specific to one particular environment
    - E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

POLITECNICO MILANO 1863

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
    - Create proper code to exercise the other methods/functions
    - Use unit test automation frameworks

Design Principle 11: Design defensively

POLITECNICO MILANO 1863

- Be careful when you trust how others will try to use a component you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check that all of the inputs to your component are valid: the preconditions
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

POLITECNICO
MILANO 1863

Architectural styles

Part 1: client server

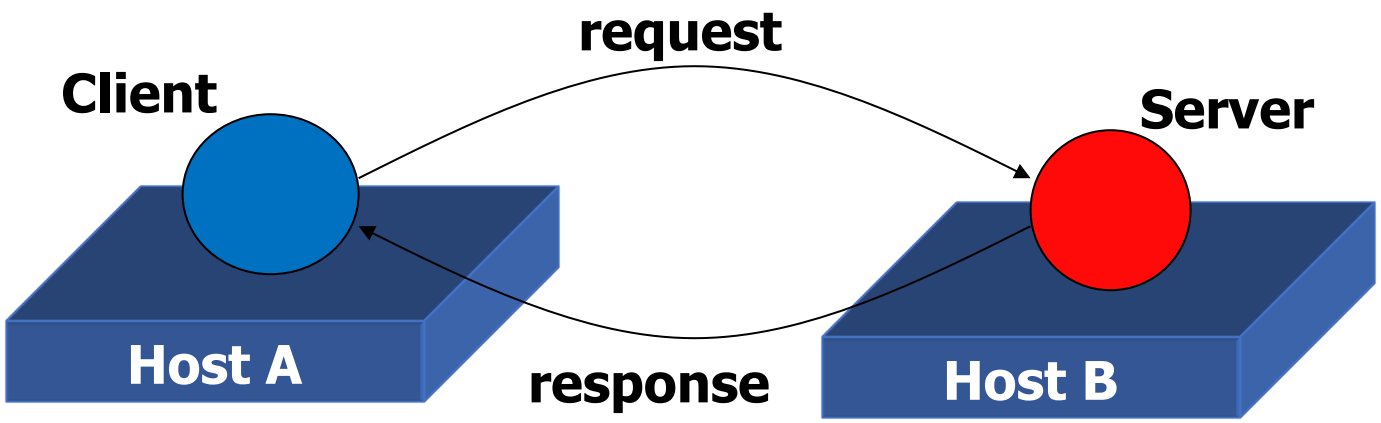
Architectural Style

POLITECNICO MILANO 1863

- “an architectural style determines the **vocabulary** of **components** and **connectors** that can be used in instances of that style, together with a set of **constraints** on how they can be combined.
  These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition.” [Garland & Shaw]

[Garland & Shaw] David Garlan and Mary Shaw, “An Introduction to Software Architecture”, Jan 1994, CMU-CS-94-166. http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

Client-server



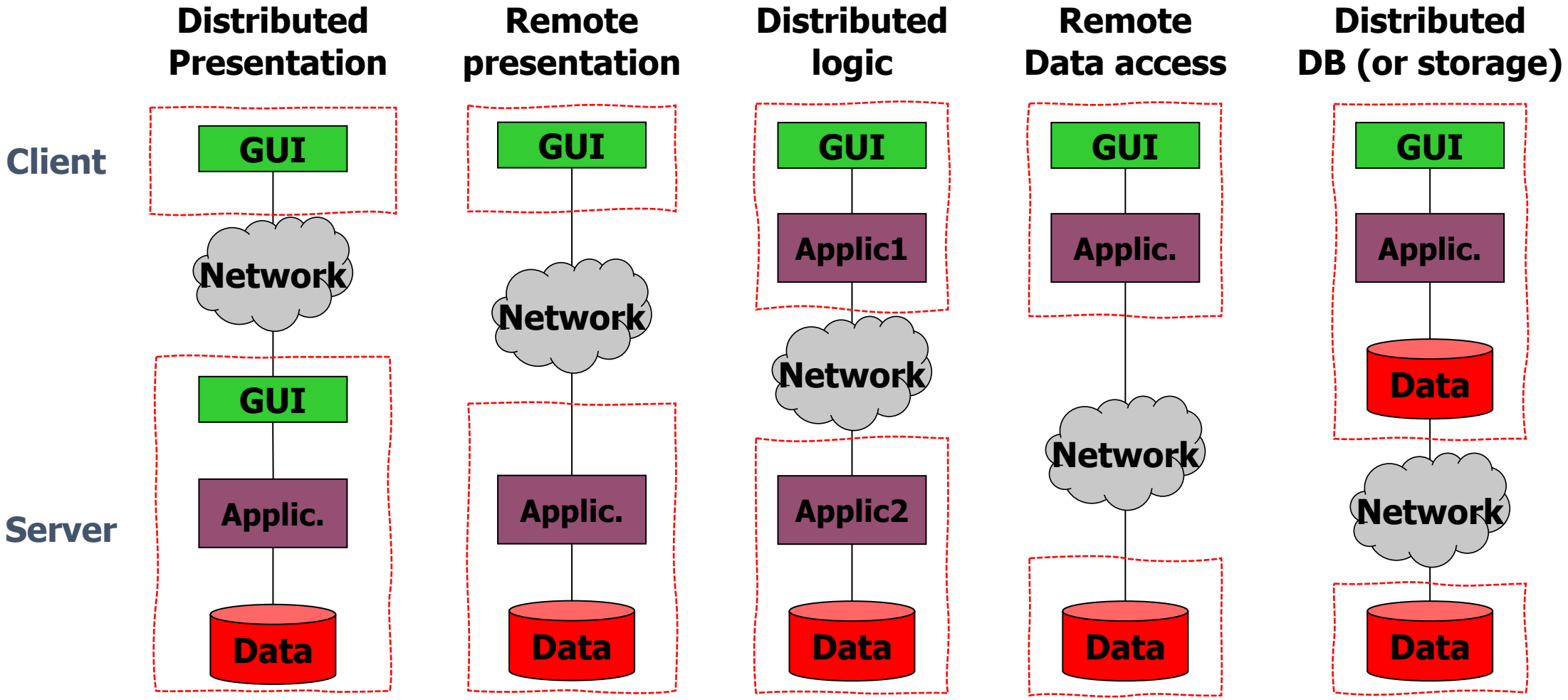
- Two component roles
  - Client issues requests
  - Server provides responses

Why do I use it?

POLITECNICO MILANO 1863

- **Multiple users** need to access a **single resource**
  - e.g., database
- There is a preexisting software we must **access remotely**
  - e.g., email server
- It is convenient to organize the system around a **shared piece of functionality** used by multiple components
  - e.g., authentication/authorization server

Organization of Client-Server software: thin vs fat clients



Client-server: main technical issues

POLITECNICO MILANO 1863

- **Design** and **document** proper interfaces for our server
- Ensure the server is able to **handle multiple simultaneous requests**
  - Forking vs thread pooling

Interface design

POLITECNICO MILANO 1863

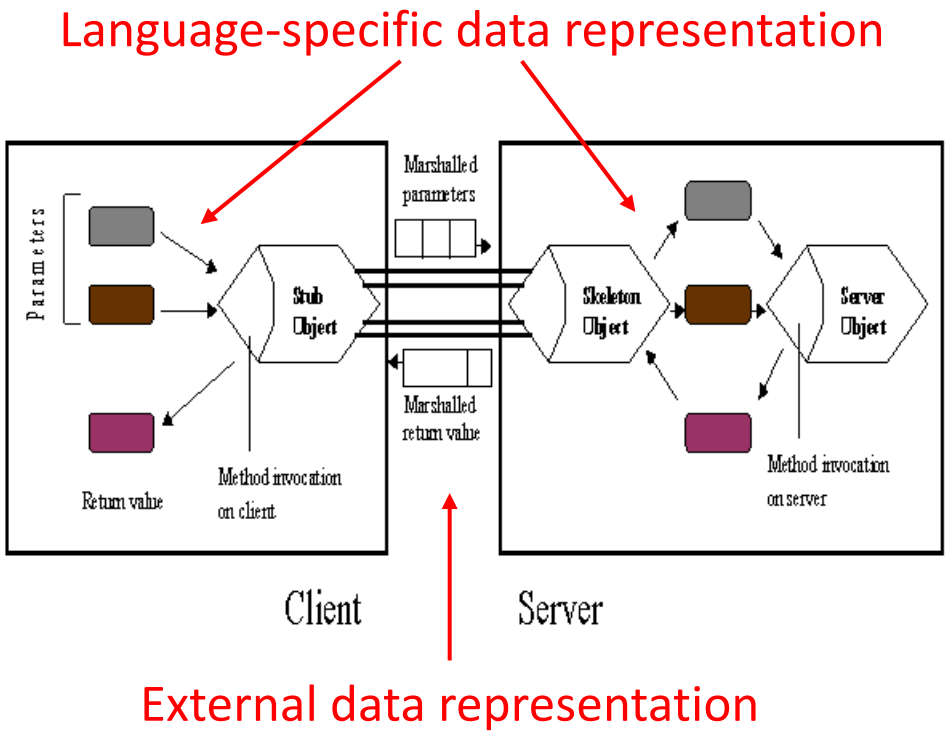
- An interface is a boundary across which components interact
- Proper definition of interfaces is an architectural concern
  - Impacts maintainability, usability, testability, performance, integrability
- Two important guiding principles
  - **Information hiding**
  - **Low coupling**
- An interface shall encapsulate a component implementation so that this can be changed without affecting other components

Aspects to consider during interface design

POLITECNICO MILANO 1863

- Contract principle: Any resource (operation, data) added to an interface implies a commitment to maintaining it
- Least surprise principle: Interfaces should behave consistently with expectations
- Small interfaces principle: Interfaces should limit the exposed resources to the minumum
- Important elements to be defined:
  - **Interaction style** (e.g., sockets, RPC, REST)
  - **Representation** and structure of exchanged data
  - **Error handling**

Representation and structure of exchanged data



- **Representation** impact on
  - Expressiveness
  - Interoperability
  - Performance
  - Transparency

Error handling

- **Examples of issues**
  - An operation is called with invalid parameters
  - The call does not return anything
    - The component cannot handle the request in the current state
    - Hardware/software errors prevent successful execution
    - Misconfiguration issues (e.g., the server is not correctly connected to the database)
- **Possible reactions**
  - Raising an exception
  - Returning an error code
  - Log the problem

Multiple interfaces and separation of concerns

POLITECNICO MILANO 1863

- A server can offer multiple interfaces at the same time
- This enables
  - Separation of concerns
  - Different levels of access rights
  - Support to interface evolution (see next)

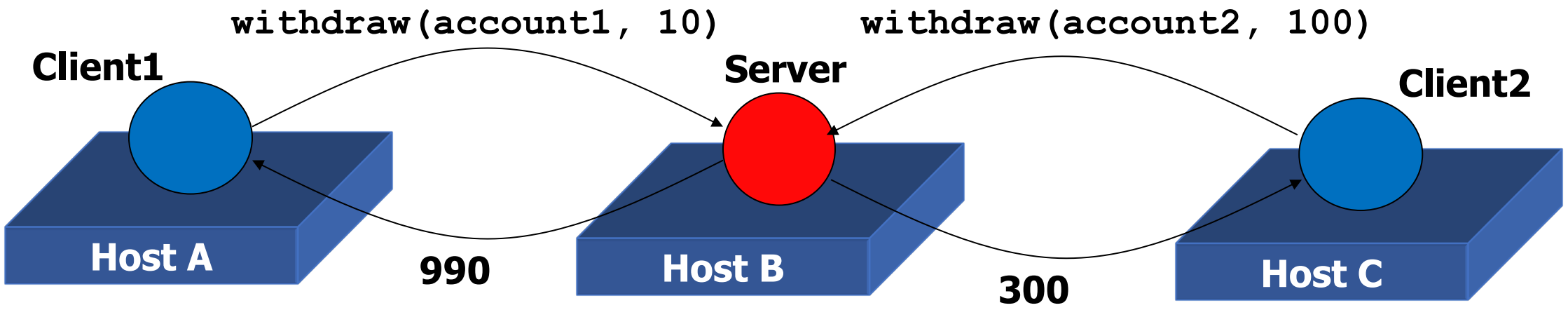
Interface evolution

POLITECNICO MILANO 1863

- Interfaces constitute the contract between servers and clients
- Sometimes interfaces need to evolve (e.g., to support new requirements)
- Strategies to support continuity
  - **Deprecation**: declare well in advance that an interface version will be eliminated by a certain date
  - **Versioning**: maintain multiple active versions of the interface
  - **Extension**: a new version extends the previous one

Handling multiple requests

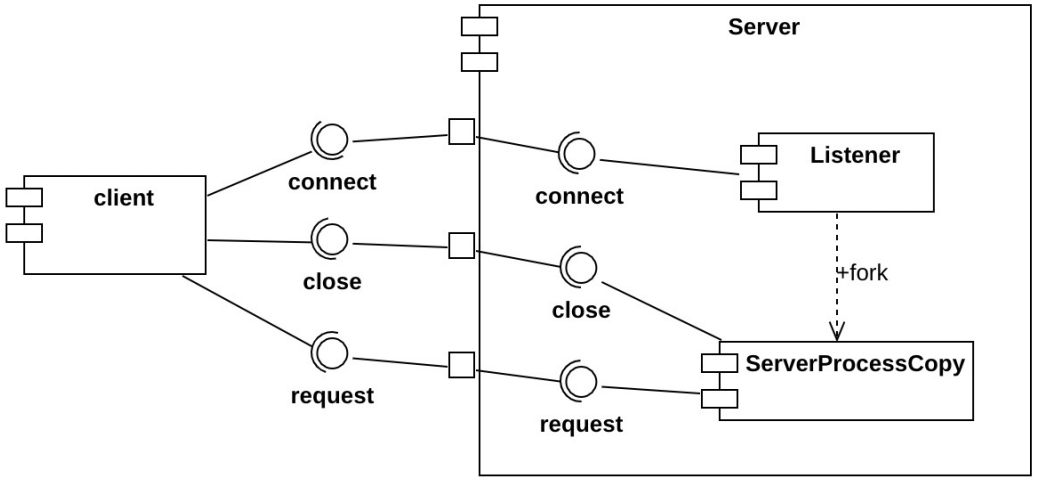
- The server must be able to receive and handle requests from multiple clients, example:

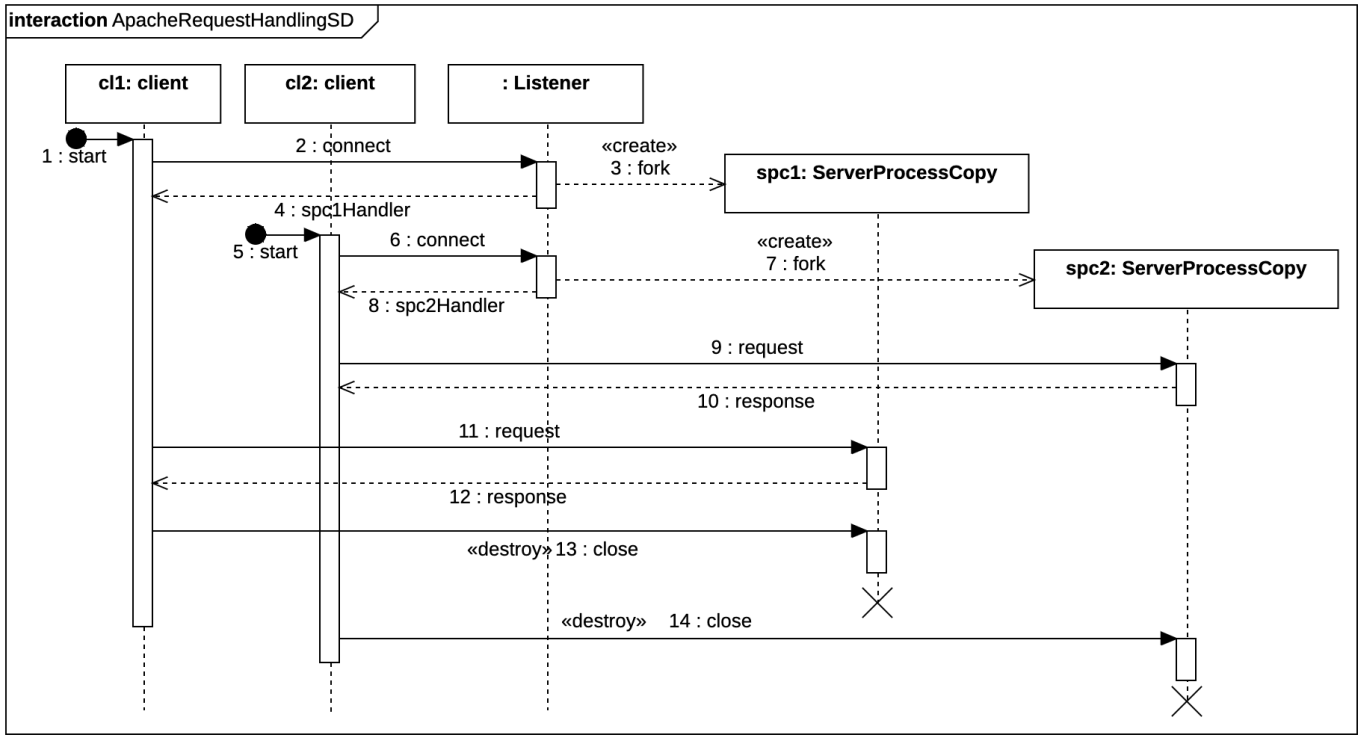


- How can the server serve the second `withdraw` if the first one is still ongoing?

Handling multiple requests: a first approach (forking)

- Simple approach used by Apache Web Server
  - **One process per request** or per client



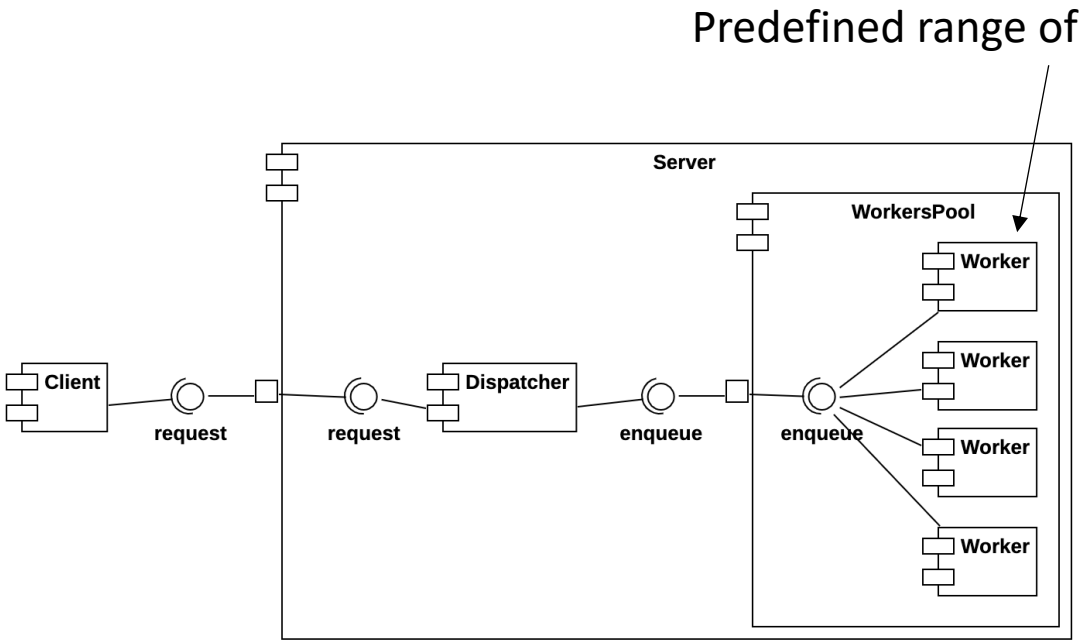


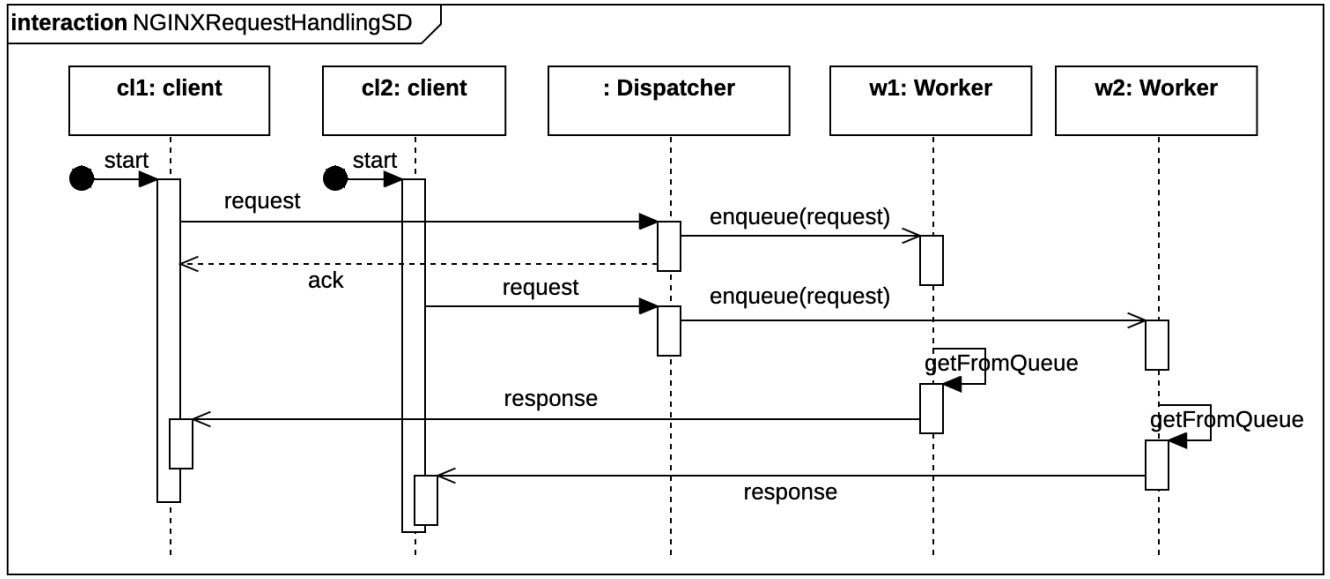
Handling multiple requests: a first approach (forking)

- Strengths
  - Architectural simplicity
  - Isolation and protection given by the “one-connection-per-process” model
    - Slow processes do not affect other incoming connections
  - Simple to program
  - Effective solution till ~2000
- Issues
  - Growth of the WWW over the past 20 years (#users and weight of web pages)
  - #active processes at time *t* difficult to predict and may saturate the resources
  - Expensive fork-kill operations for each incoming connection
  - → **Scalability issues!**

Handling multiple requests: alternative approach (worker pooling)

- Alternative approach adopted by NGINX Web Server
- Designed for high concurrency — deals with scalability issues





Handling multiple requests: alternative approach (worker pooling)

- Note (1): NGINX tackles the previous issues by introducing a new **architectural tactic**
  - **Tactic = Design decisions that influence the control of one or more quality attributes**
- Strengths of the new approach
  - Number of workers is fixed → they do not saturate available resources
  - Each worker has a queue
  - When queues are full the dispatcher drops the incoming requests to keep high performance
  - Dispatcher balances the workload among available workers according to specific policies

Handling multiple requests: alternative approach (worker pooling)

- Note (2): architectural tactics introduce **quality attribute trade-offs**

- NGINX decided to **optimize scalability** and **performance** by **sacrificing availability (in some cases)**
  - When all worker queues are full, the dispatcher drops incoming requests rather than spawning new workers