Introduction to Software Engineering

Software architecture

Philippe Lalanda

Philippe.lalanda@imag.fr

http://membres-liglab.imag.fr/lalanda/

Outline

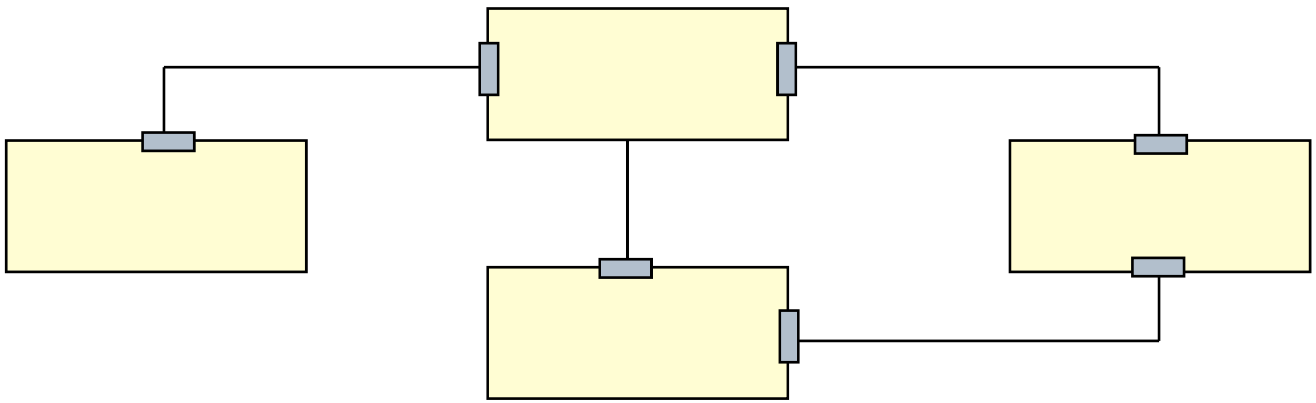
- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Reminder: two levels of design

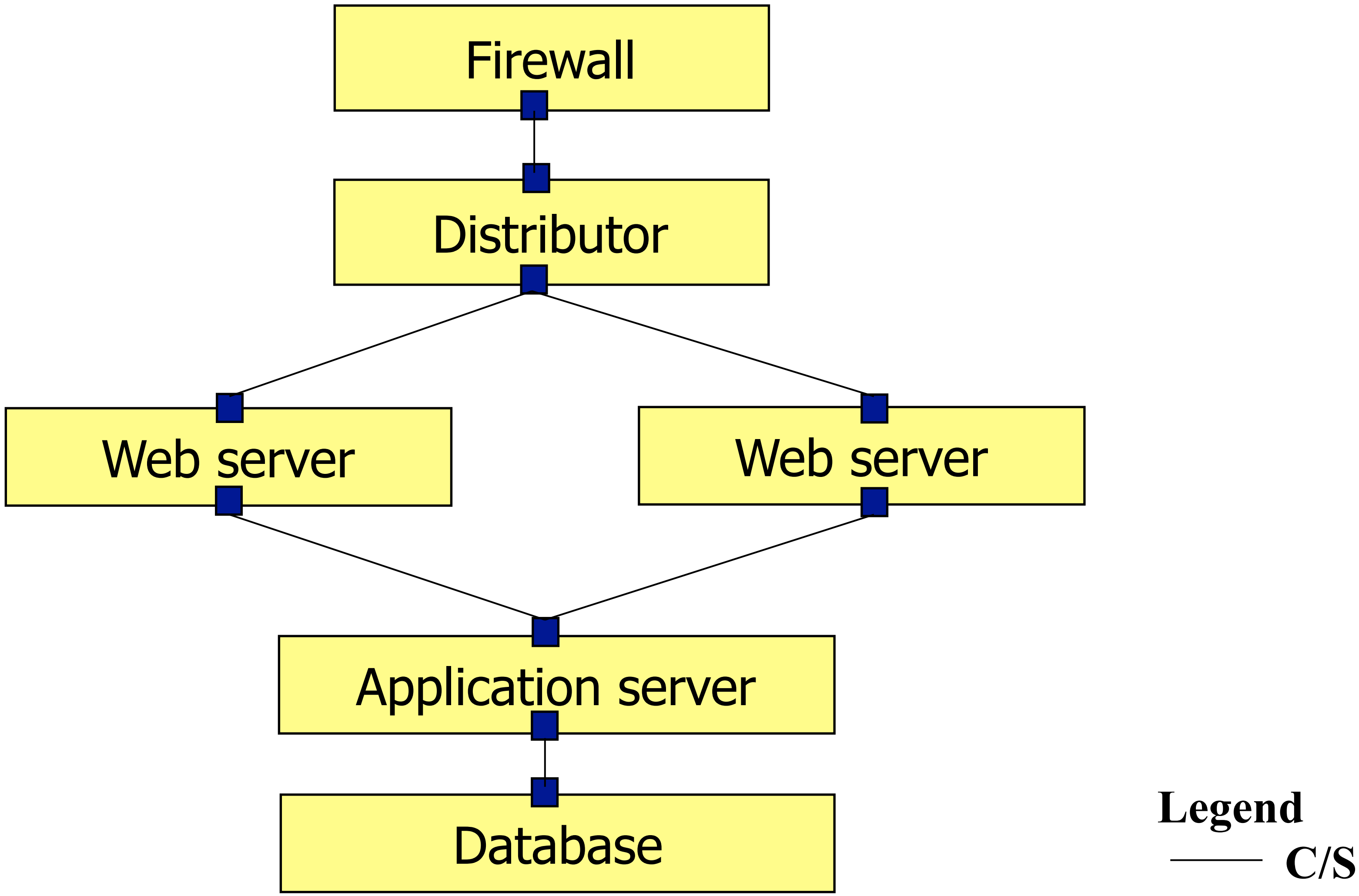
- Global design
  - Software architecture
    - Components
    - Connectors
- Detailed design
  - Programming concepts
    - Objects, structured types, …
    - Methods, procedures, …

Software architecture

- An <u>abstract</u> specification expressed as <u>components</u> interacting through <u>connectors</u>
  - Its purpose is not to be executable – it's a model
  - It is a decomposition that has to meet requirements



Example

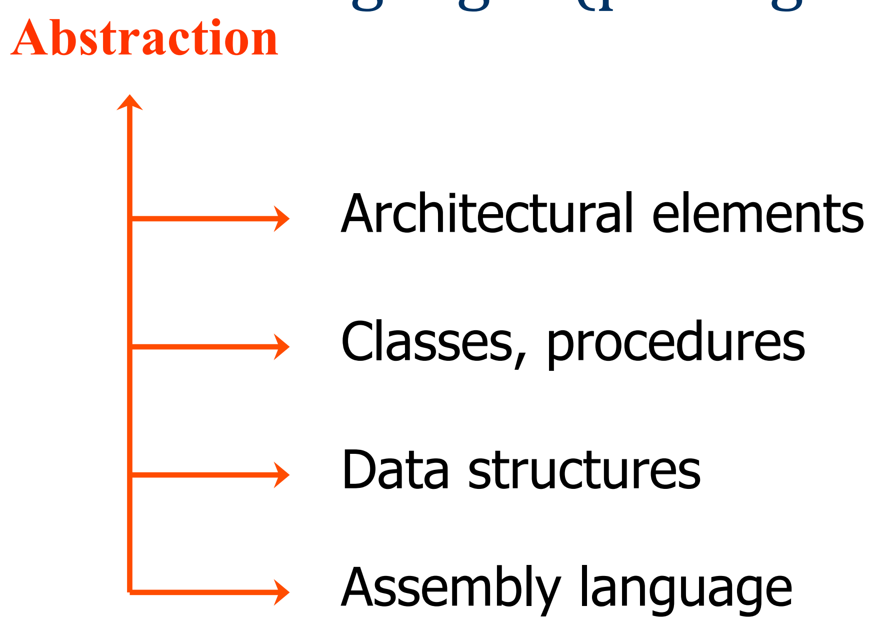

Firewall
Distributor
Web server
Web server
Application server
Database
Legend
C/S

Software architecture - specification

- An architecture defines the overall organization (structure) of the code to come
  - It is a guideline for designers / developers
- It must be precise
  - Choices are made
  - Not a (fuzzy) functional decomposition
- It is complete
  - Gives all necessary information for detailed design
    - Possibly performed by partners, remote teams, …

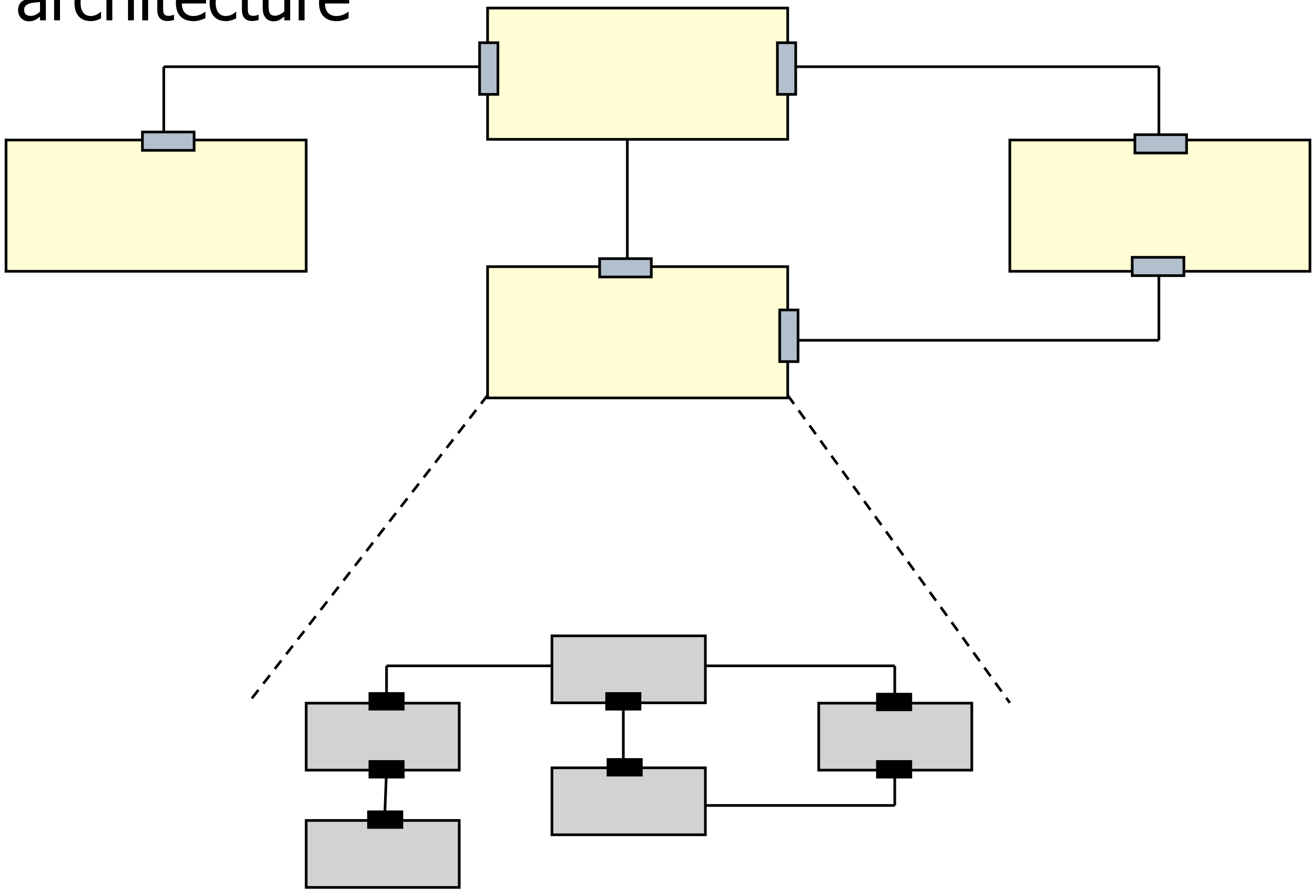
Software architecture - abstraction

- An additional level of abstraction
  - It does not bother with implementation details
  - It defines relevant properties of structuring elements
  - Not existing in current languages (packages ...)



Software architecture - abstractions

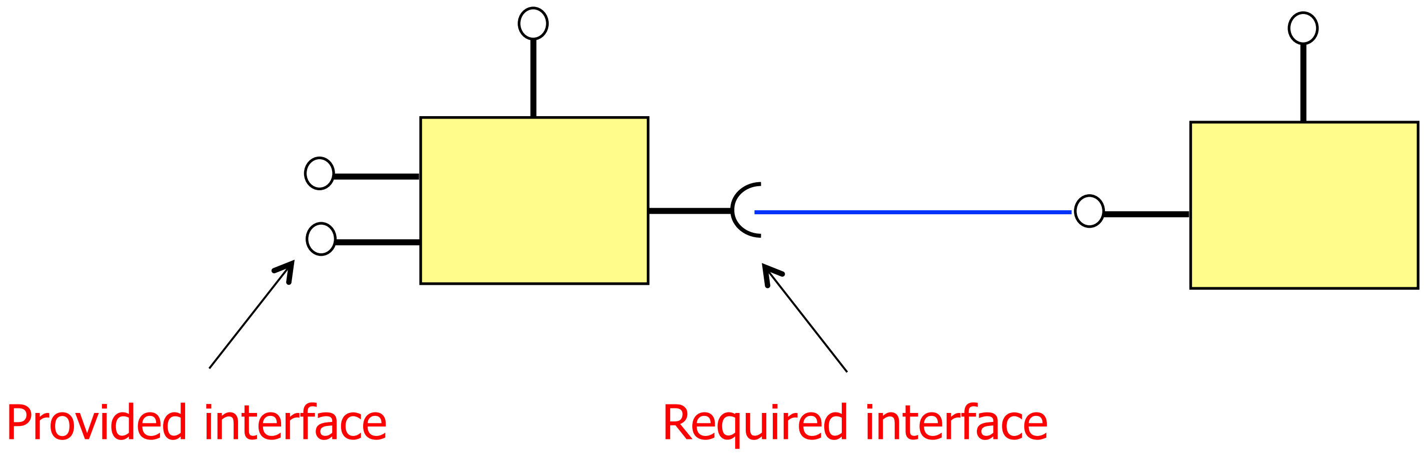
Global architecture



Architecture of an element

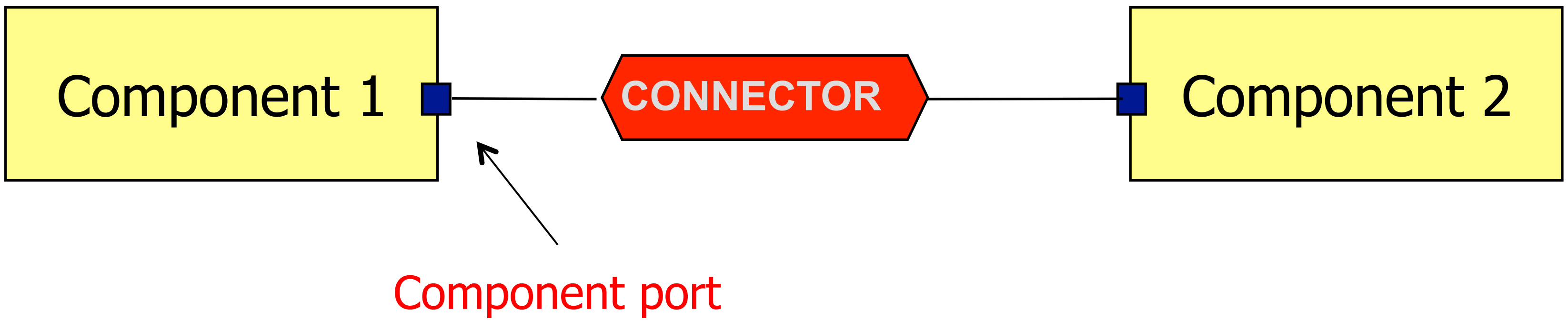
Software architecture - components

- Functional units specifications
  - Clearly defined, coherent, comprehensible
  - With functional dependencies
  - With properties, constraints

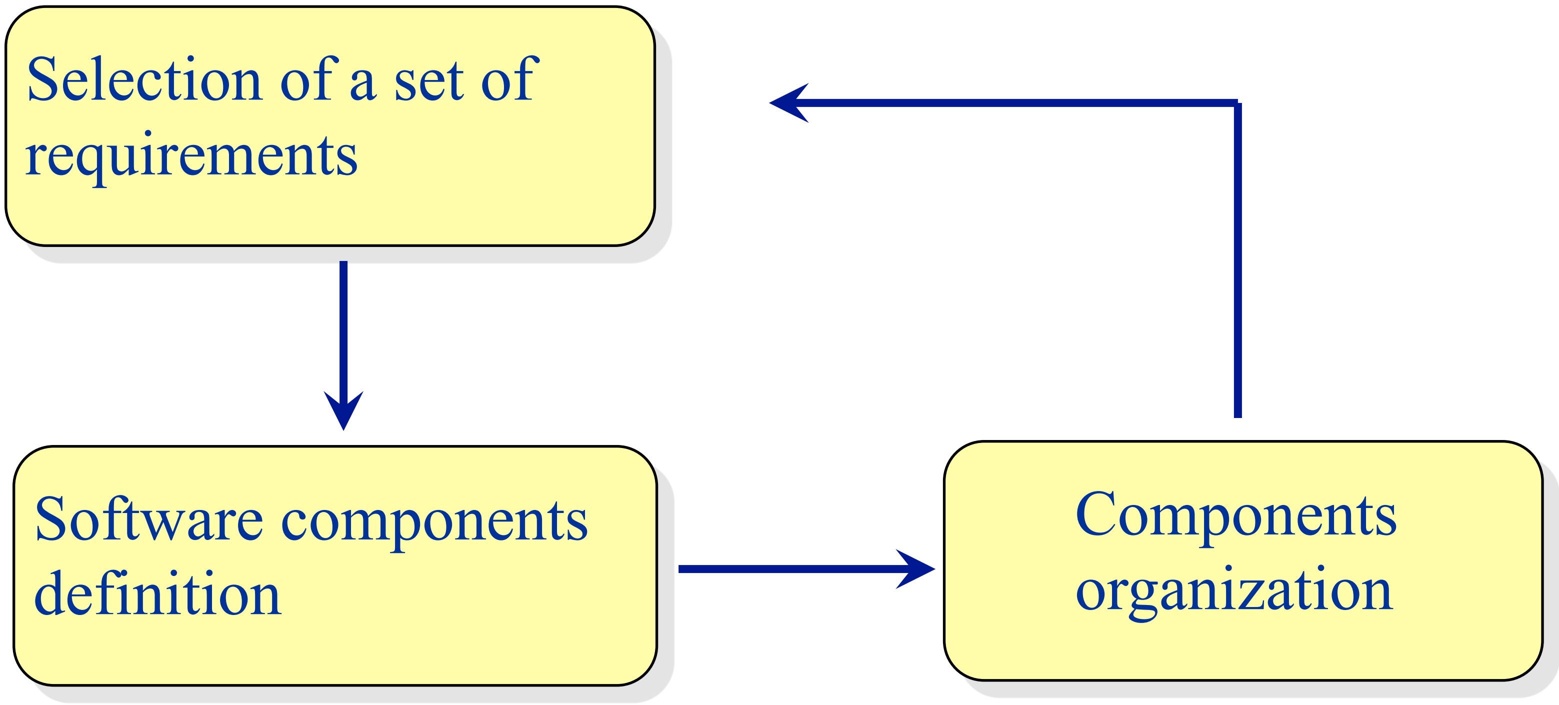


Software architecture - connectors

- First order objects
  - Defines interactions between components
  - Can be pretty complex
  - No specification language yet



Designing software architecture



Design challenge

- Selection of requirements
  - Find out relevant requirements at each iteration
- Definition/refinement of components
  - Driven by functional requirements and by the design context
  - Some are techniques, others are business specific
- Organization of components
  - Interaction patterns, rules, ...
  - Driven by non functional requirements

Architectural styles

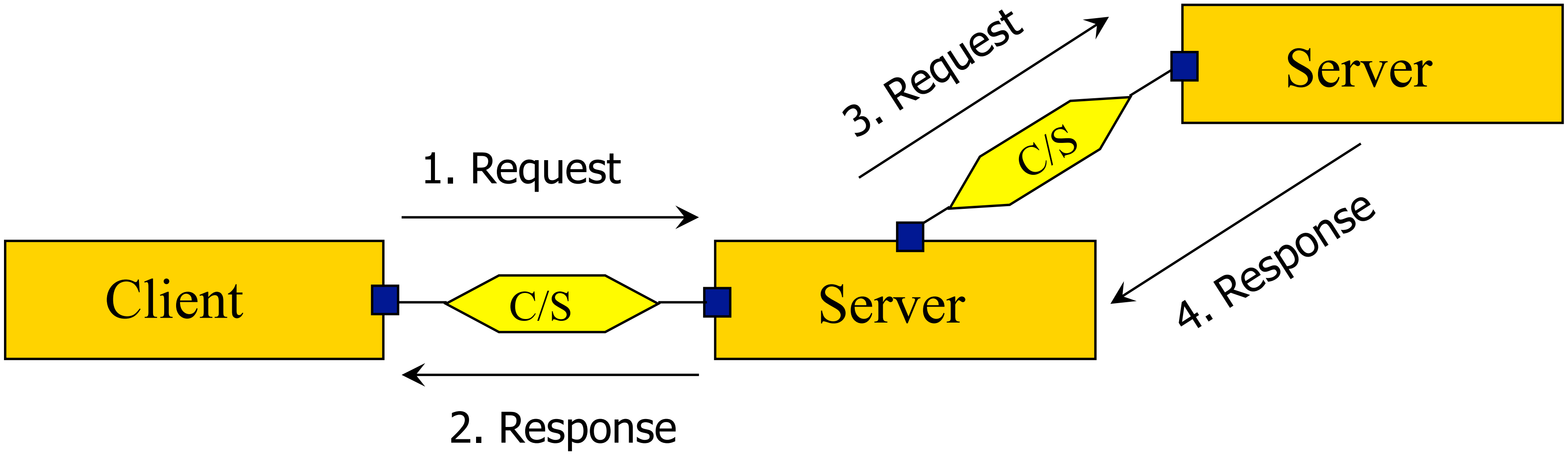
- Idea
  - Styles used in a recurrent way in successful systems should be reused
- Principles
  - Describe successful styles (books)
  - Learn them
  - Combine them to meet your project goals

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Client / server style

- Structure the system into components interacting through well defined functional interfaces
  - Synchronous interactions, initiated by the client
  - Interactions are deterministic and non continuous

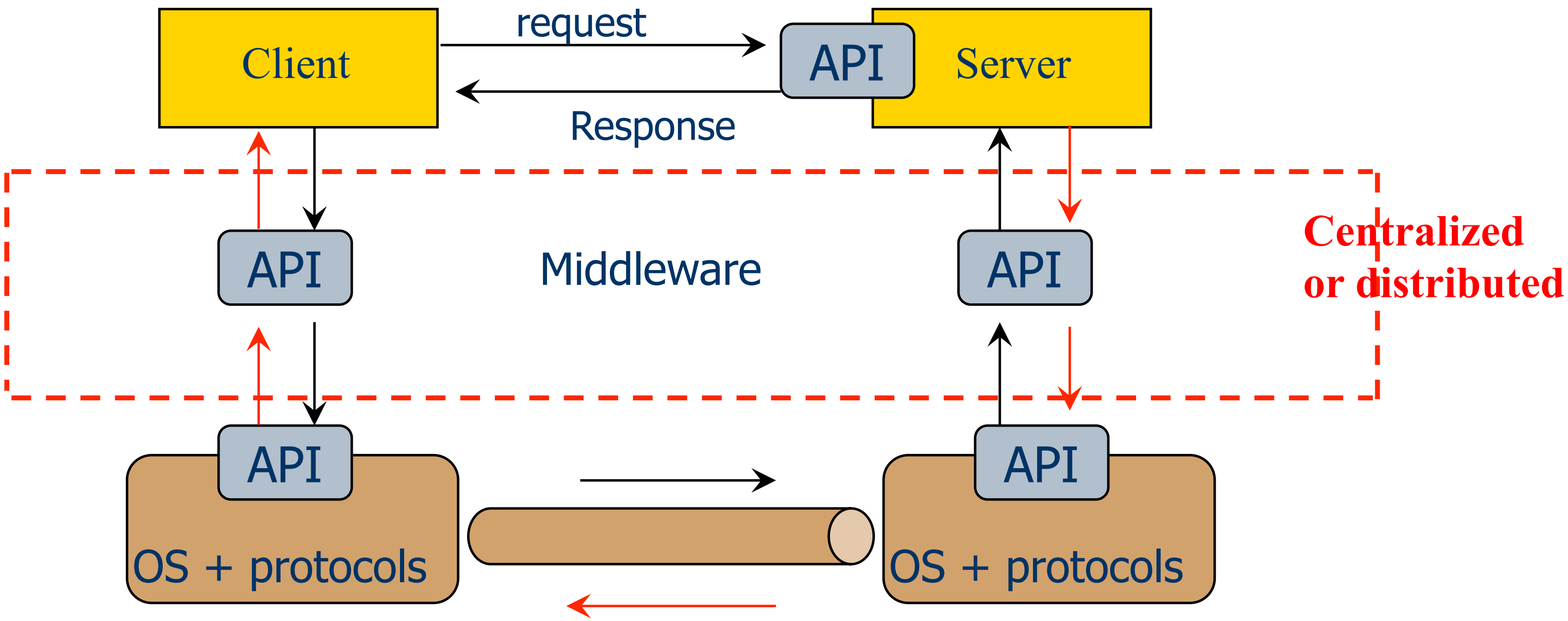


Style characterization

Elements	❑ Clients: service requesters ❑ Servers: service providers ❑ Ports: functional interfaces ❑ Connectors: (R)PC calls
Computing model	❑ Communication is initiated by clients. ❑ They ask for a service a wait for the answer.
Constraints	No constraints on topology. Possible constraints: - limited number of clients - service scheduling

Implementation

- Use a middleware
  - To handle communication (mainly when distributed)
  - Can hide localizations, initiate / stop interactions, ...



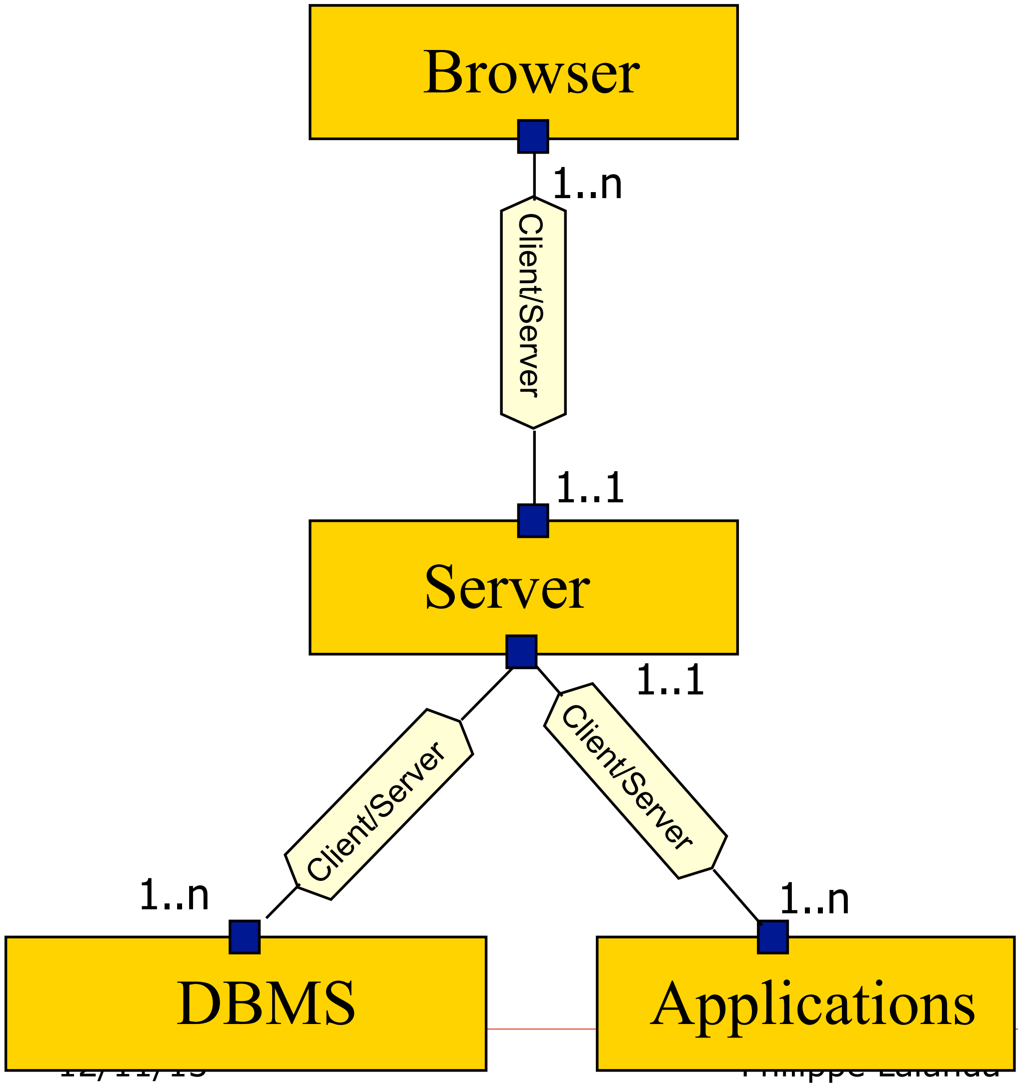
Advantages

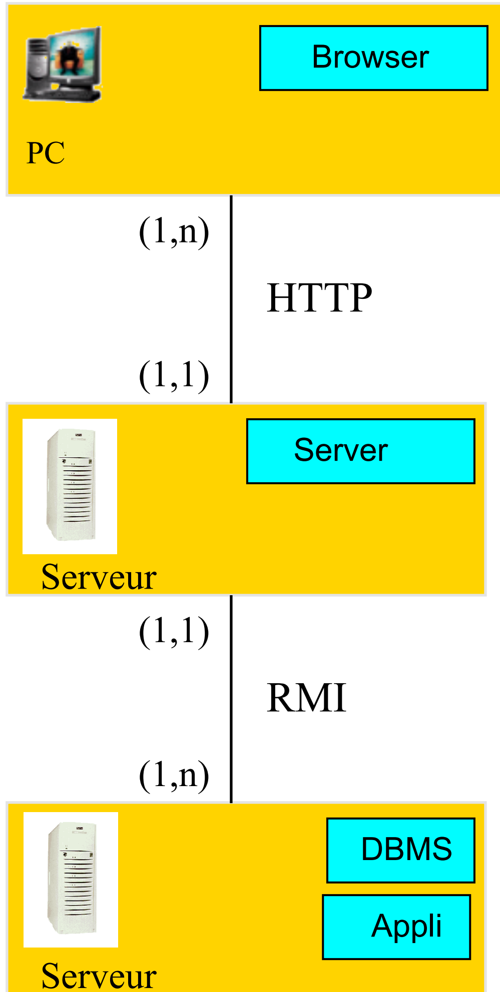
- Simple and clear
  - Easy to communicate
- Good SE practices
  - Strongly types APIs
    - Safer !
  - Encapsulation and information masking
    - Coherence and modularity
- Should favor test, debug, evolution

Limits

- Performance
  - Communication cost
  - Several services might be called to meet a given goal
  - Scalable
- Reusability
  - Components dependencies
  - Non functional code in the components
- Distribution is complex
  - A middleware is rapidly necessary

Example: Web-based systems





Conclusion

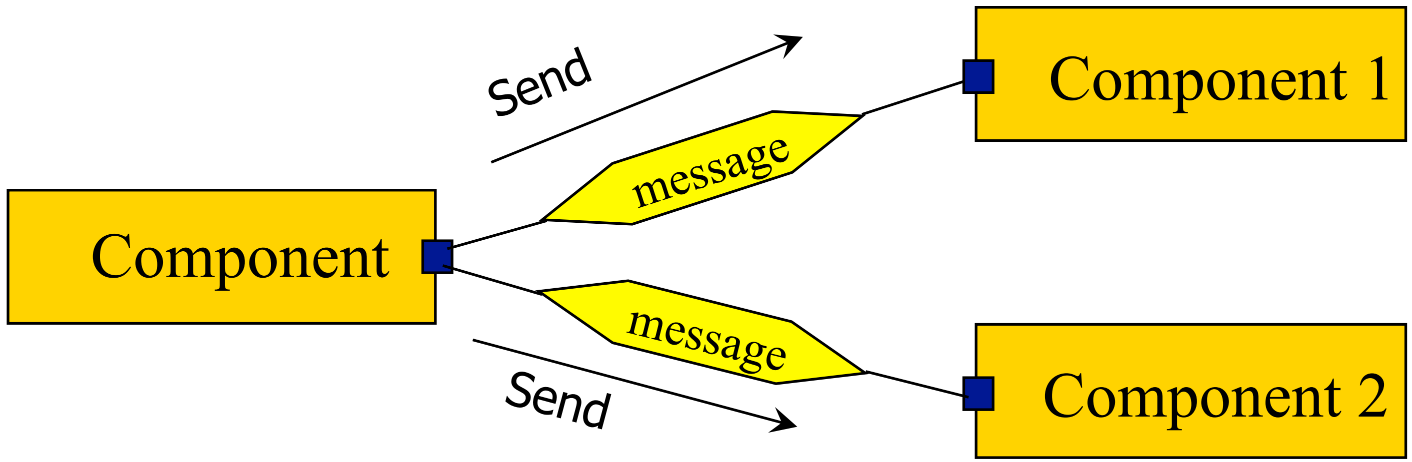
- This is "classic" style which is used in many projects
  - Corresponds somehow to the state of the art in SE
  - Towards Component based software engineering
- Use it when
  - No heavy constraints on performance
  - APIs are known and can be typed
  - No massive data flow

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Message oriented style

- Structure the system into components interacting through messages
  - Asynchronous interactions, initiated by the emitter
  - Interactions are deterministic and non continuous



Style characterization

Elements

- Emitters components: message senders
- Receivers components: message consumers
- Ports: technical interfaces (*send, receive*)
- Connectors: message transporters

Computing model

- Communication is initiated by emitters.
- Messages are consumed at consumers speed.

Constraints

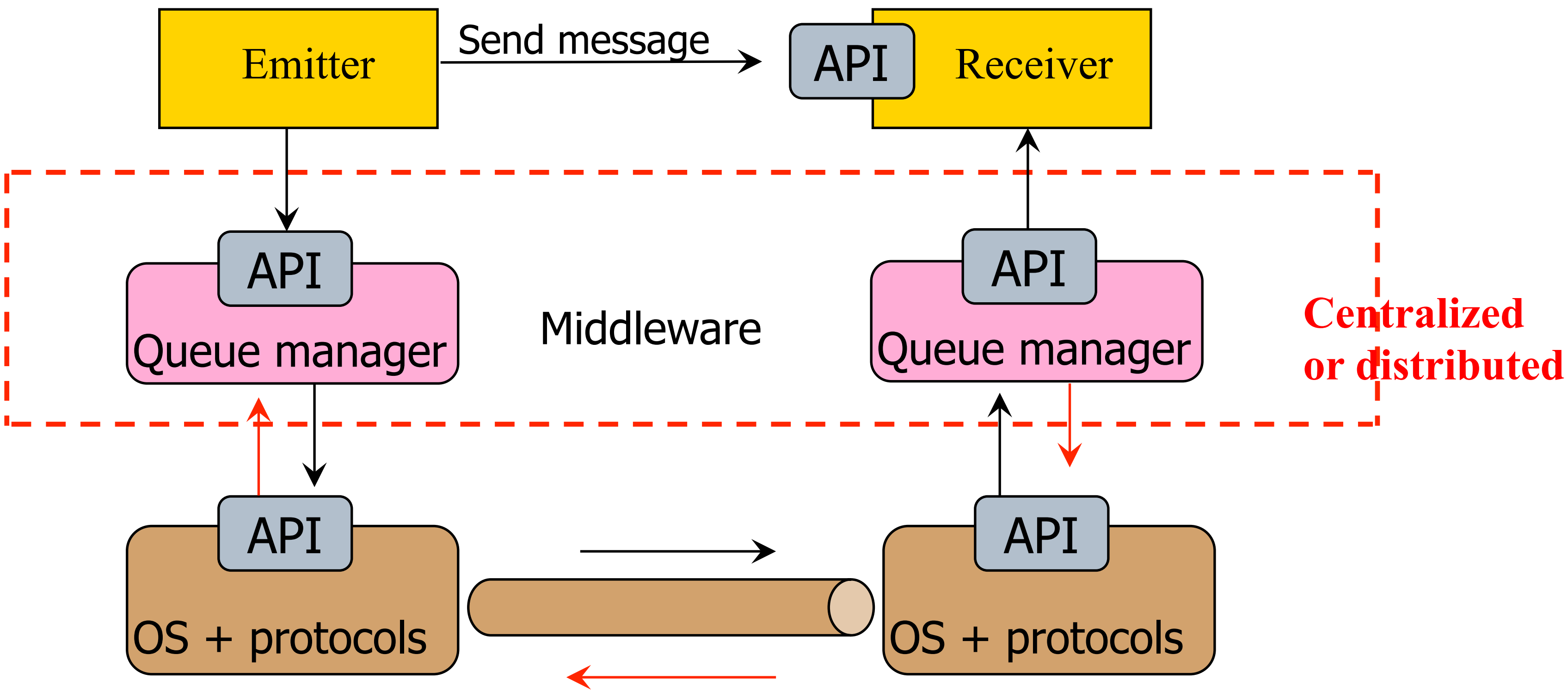
No constraints on topology.

Possible constraints:

- limited number of messages can be received
- message typing

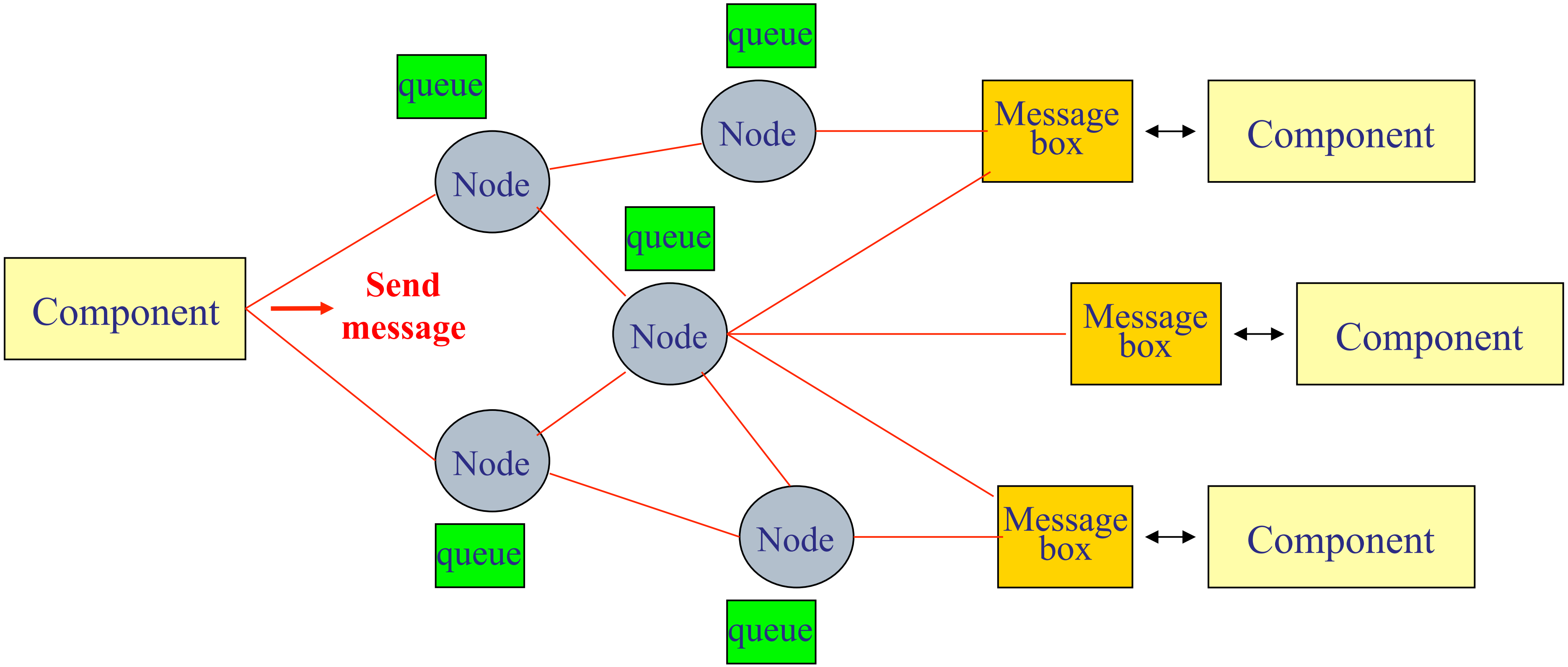
Implementation

- Use a middleware
  - Handle communication and message management



Implementation – typical architecture

- Distributed, scalable context



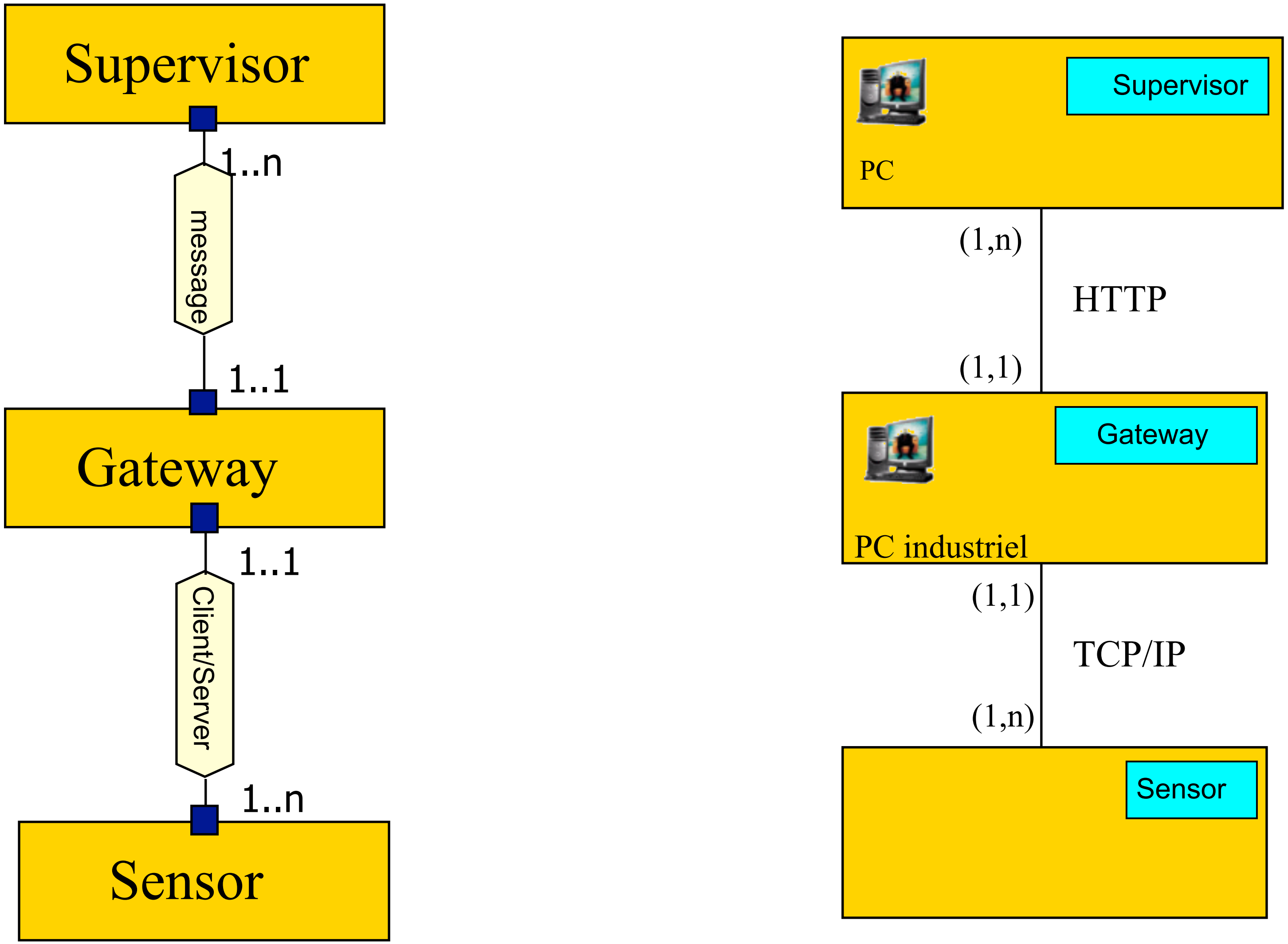
Advantages

- Relatively clear
  - Easy to communicate in simple cases
- Efficient
  - Only technical interfaces
  - No mandatory typing
  - Communications can be limited
    - Several kinds of information can be sent in a unique message
- Decoupling
  - Weak knowledge about receivers

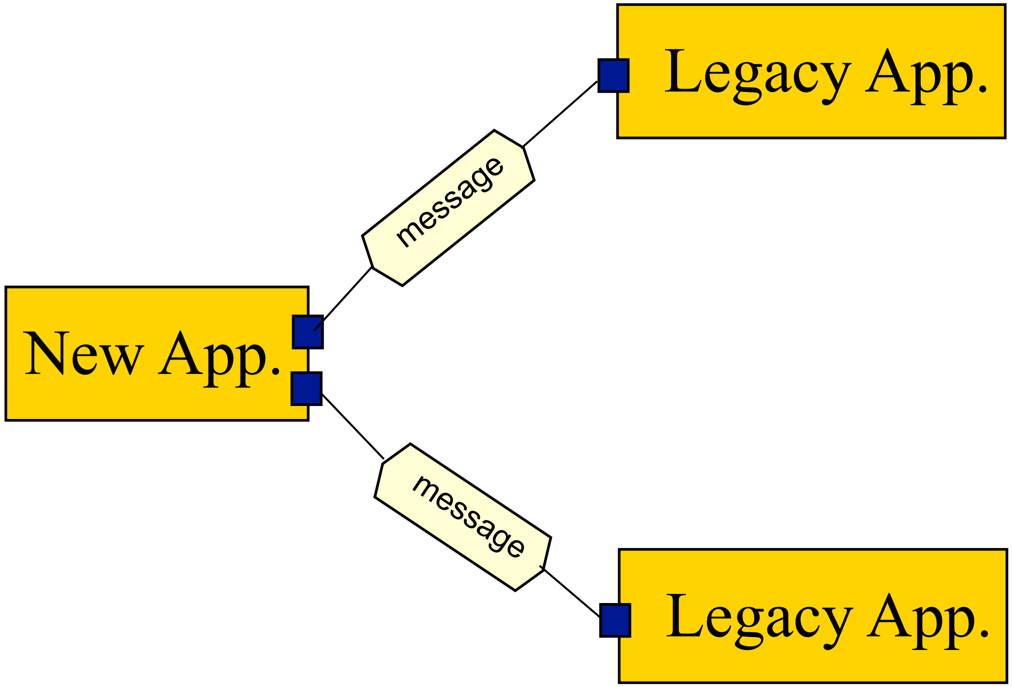
Limits

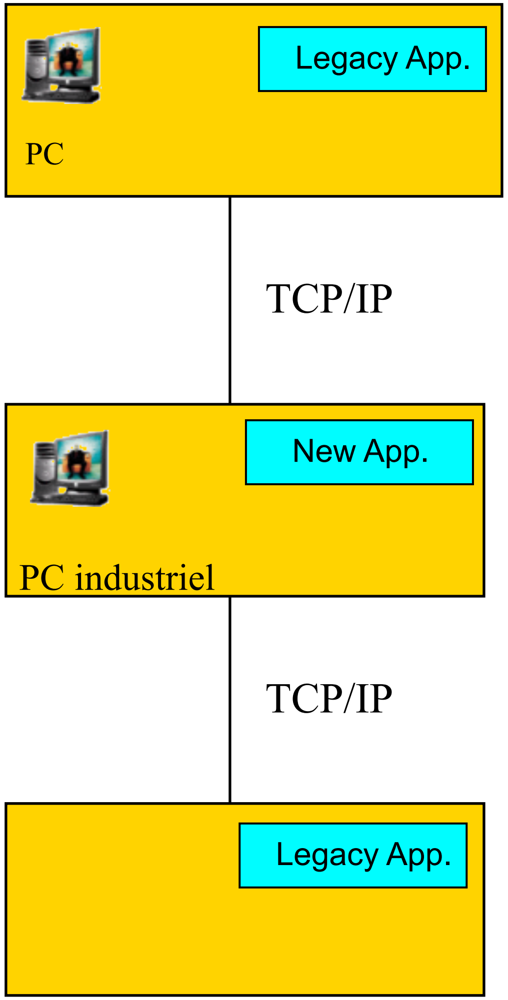
- May be unclear
  - Communications are not always explicit – when using multicast for instance
- Weak regarding SE
  - No strong typing
  - Hard to develop, test and debug
- Message management can become complex
  - Message persistency, fault handling, …
  - A middleware is necessary

Example: data collection



Example: application integration





Conclusion

- This is a very popular and useful style
  - Old and pragmatic
  - Not perfect in term of SE

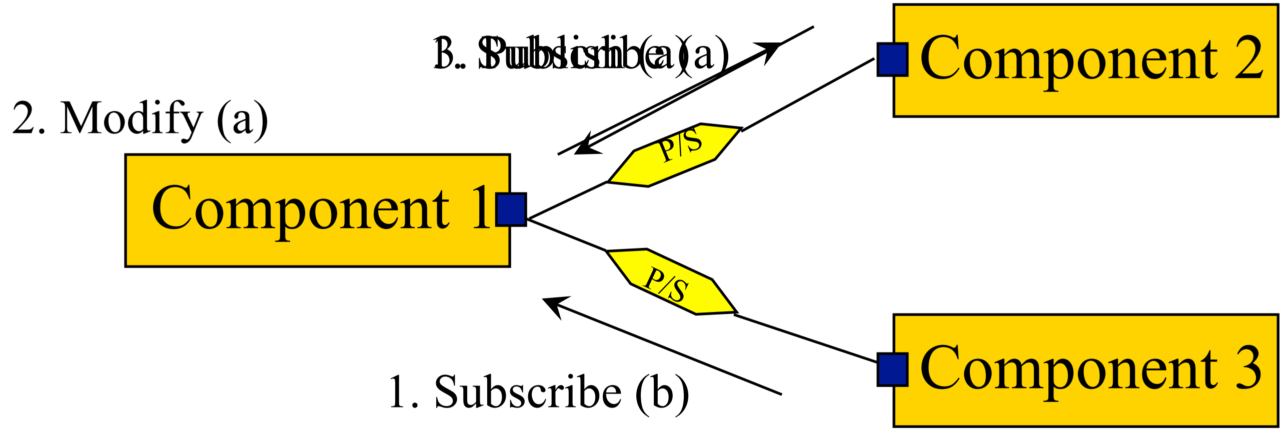
- Use it when
  - Constraint on performance (not too much though!)
  - Integration of applications with no clear APIs
  - Asynchronous needs (data collection, unknown frequency rate, ...)

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Publish / subscribe style

- Structure the system into components interacting through messages with <u>subscription</u>
  - Asynchronous interactions, initiated by the emitter
  - Based on subscriptions on topics
  - Interactions are deterministic and non continuous



Style characterization

Elements

- Publishers: message senders
- Subscribers: message consumers
- Ports: technical interfaces (*send, receive, subscription management*)
- Connectors: message transporters

Computing model

- Communication is initiated by emitters.
- Communication only concerns subscribers
- Messages are consumed at consumers speed.

Constraints

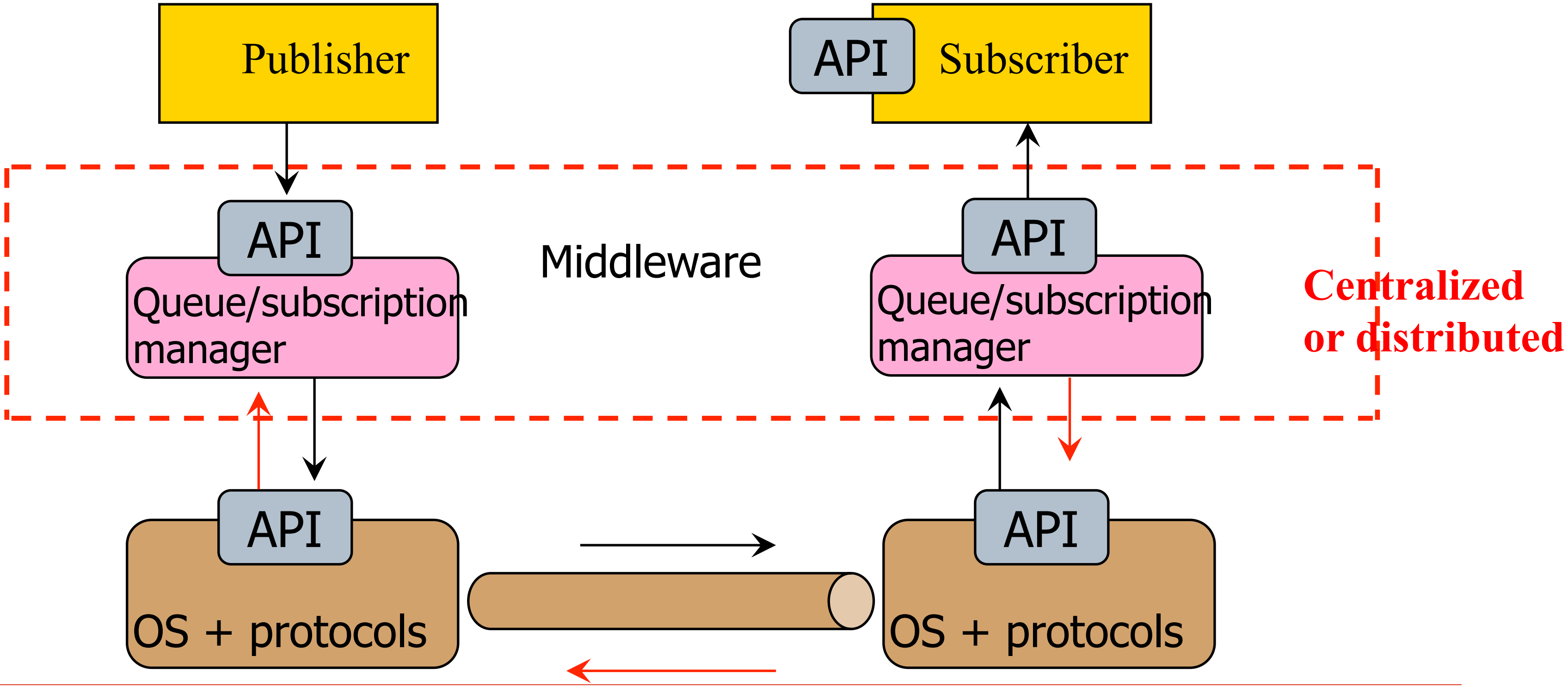
No constraints on topology.

Possible constraints:

- limited number of subscribers
- limited number of topics

Implementation

- Use a middleware
  - Handle communication, message management and subscription management



Implementation issues

- Interfaces to receive data
  - One interface per topic: when a components subscribes to a topic, it provides a handle
  - A single interface for any topic: more analysis work
  - Taken in charge by the middleware

- Subscription
  - To a given component: providers are known
  - To a topic: request sent to the middleware

Implementation issues – some more

- Issues to be treated
  - Dynamic creation of topics
  - Priorities on topics
  - Different levels of dependabilities
    - Message sending, persistence, …

- Implementing such a middleware is a challenging task

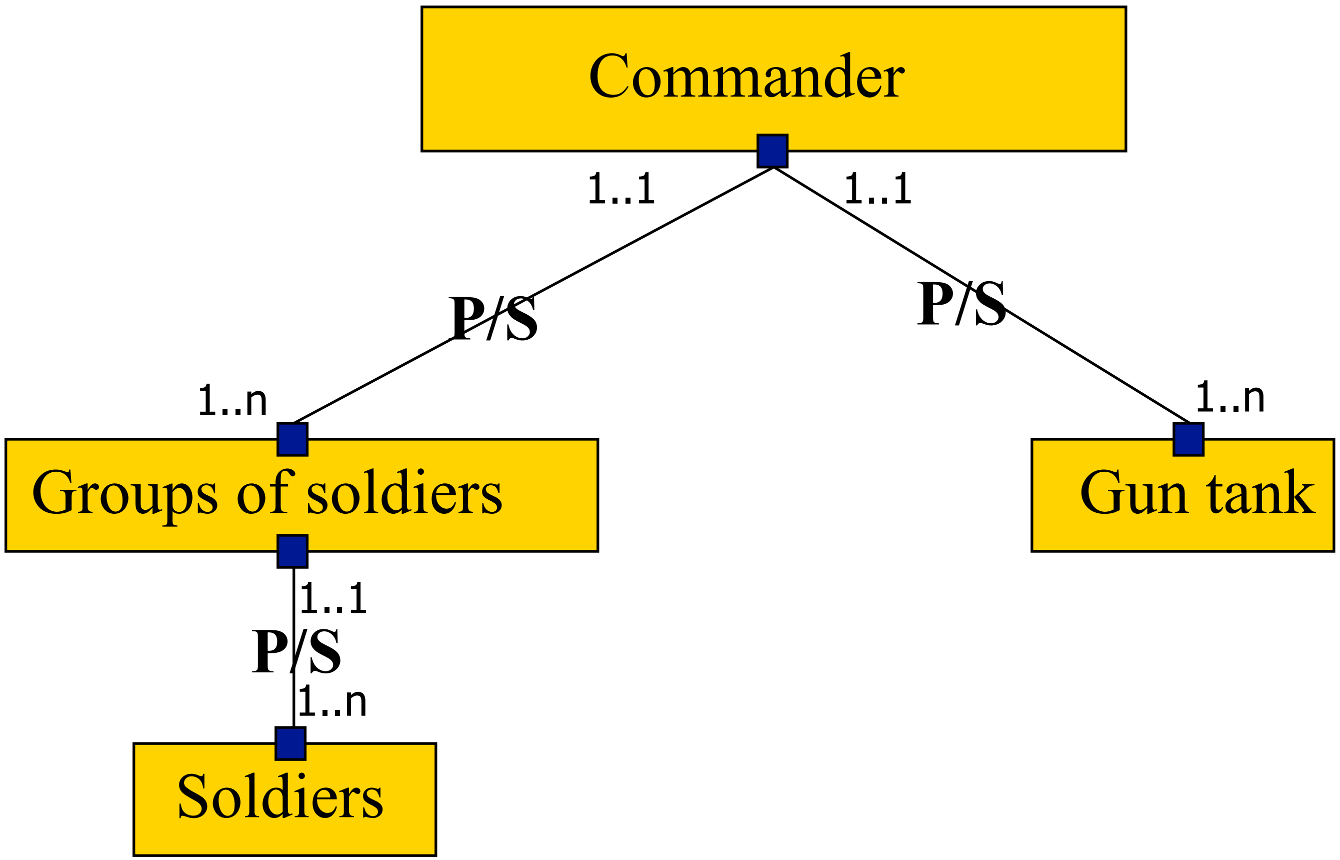
Advantages

- Efficient
  - Only technical interfaces
  - No mandatory typing
  - Communications are limited
    - Only interested parties receive messages
- Decoupling
  - Links between components are dynamic (it is a data structure coupling)

Limits

- Unclear, hard to understand
  - Communications are dynamic, not explicit
- Weak regarding SE
  - No strong typing
  - Hard to develop, test and debug
- Message management is complex
  - Subscription and topics management
  - Communication management: distribution, persistence, scheduling, …
  - Middleware definitively necessary

Example: HLA (large-scale simulation)



Conclusion

- This is a recent style, getting popular
  - Depends on the existence of appropriate middleware

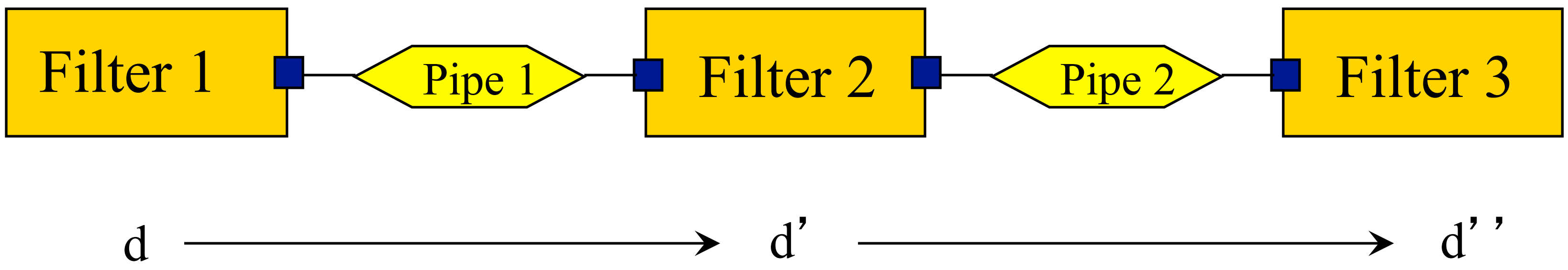
- Use it when
  - Unknown number of components
  - Dynamic interactions patterns
    - Subscriptions evolve wrt context
    - Topics may evolve too
  - Need to decouple providers and consumers

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Pipe & Filter style

- Structure the system into components interacting through data flows
  - Asynchronous interactions, initiated by the emitter
  - Interactions are deterministic and continuous



Style characterization

Elements

- Filters: components transforming data
- Ports: *in* and *out* (several per components)
- Pipes: connectors transporting data flows

Computing model

- Filters transform data coming through the *in* pipes and write them in the *out* pipes.
- Politics of data consumption may vary and may be complex

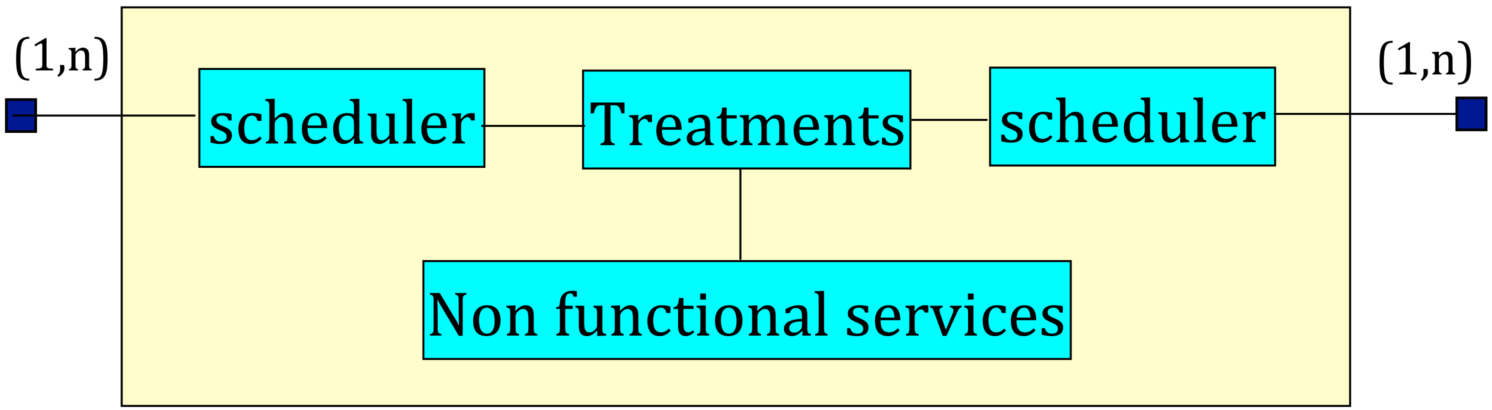
Constraints

No constraints on the way filters are connected (with pipes). Possible constraints:

- forbid cyclic topology

Implementation

- Filters may be complex to implement.
- A framework can be used to
  - Synchronize inputs
  - Define consumption strategies
  - Synchronize outputs



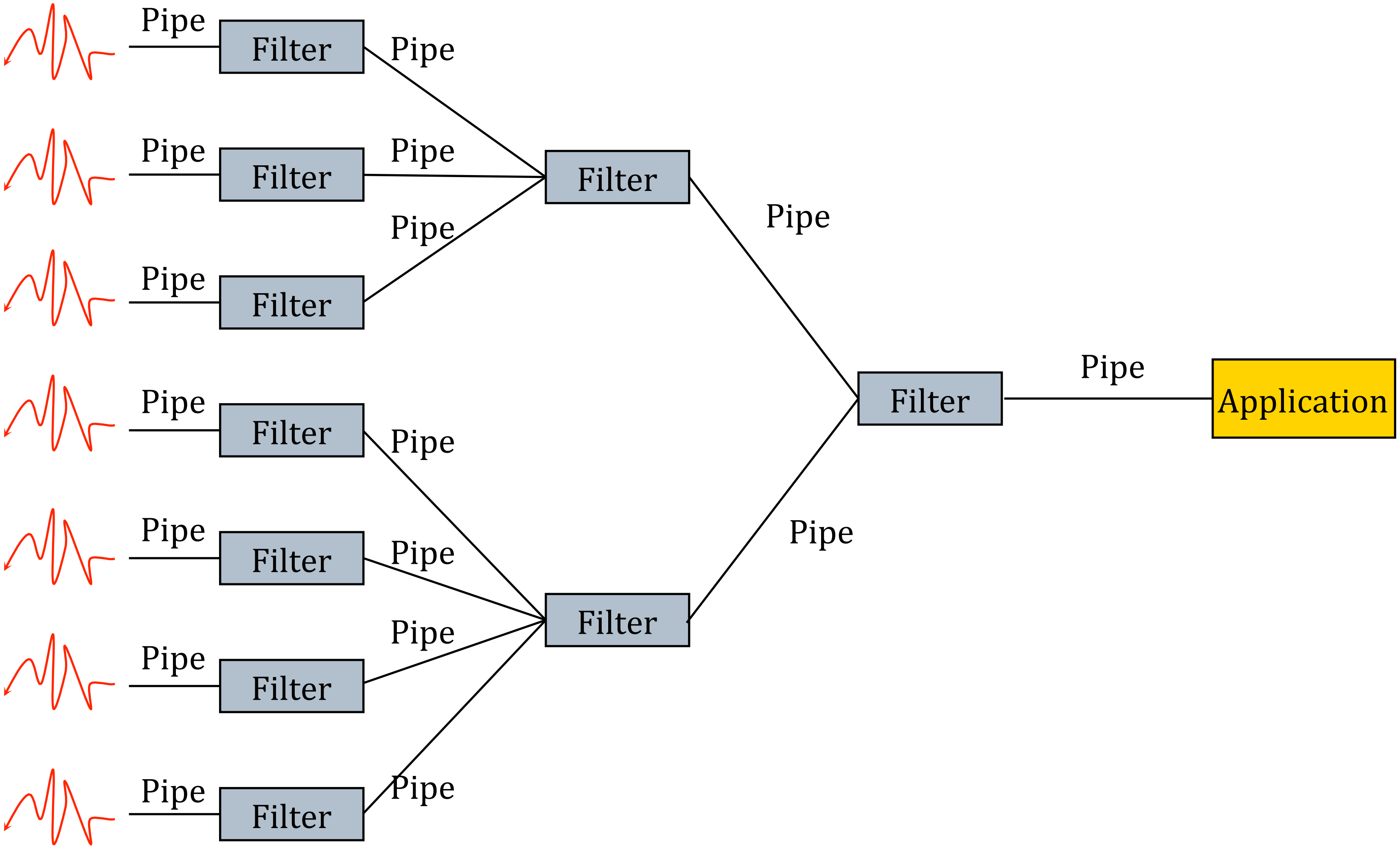
Advantages

- Simple and clear
  - Only two concepts – relatively easy to understand
- Efficient
  - No intermediary file (it is really continuous flows of data)
  - Parallel computing is possible
- Flexibility
  - Filters can be improved incrementally
  - Filters can be changed, new chains can be created
  - Good for "try and see"

Limits

- No global information is maintained
  - Error management is complex
  - Hard to debug, evaluate, ...
- Gains with parallelism are sometimes illusive
  - Pipes can cost more than filters (for distributed systems)
  - Synchronization and latency problems
- Reuse
  - Uniform data format is needed to reuse, replace filters in a flexible manner

Example: signal analysis



Conclusion

- Very popular in businesses focused on data processing
  - Tools suites have been developed
    - Build, evaluate, rearrange pipe and filters compositions

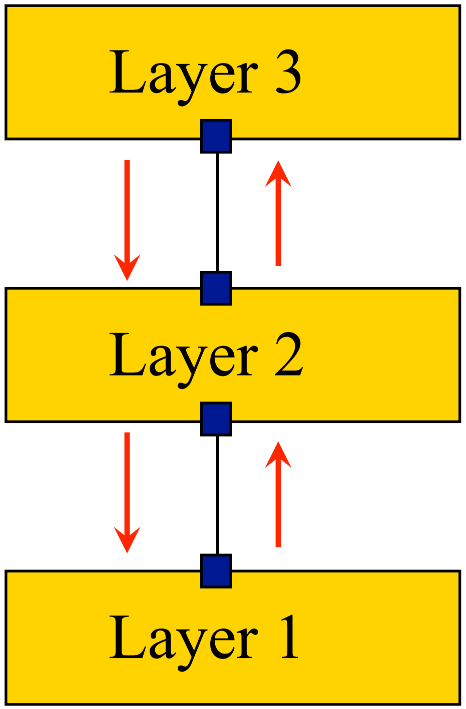
- Use it when
  - Performance is needed
  - Architectures are hard to define from the beginning
    - flexibility

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style
- Conclusion

Layered style

- Structure the system into layers (components)
  - Communication is limited to adjacent layers
  - No constraint on communication type
    - C/S
    - Message
    - Stream, ...
  - No constraints on control
    - Push vs. pull



Style characterization

Elements

- Layers (components)
- Ports: technical or business interfaces
- Connectors: no specification

Computing model

- Communication made through adjacent layers.

Constraints

No constraints on communication types

No constraints on control

Advantages

- Simple and clear
  - Communications are limited
- Favor good structuration
  - Abstraction level are naturally concretized
- Favor standardization
  - Reuse, development of a COTS market
- If used with a C/S style, it can lead to excellent SE practices!

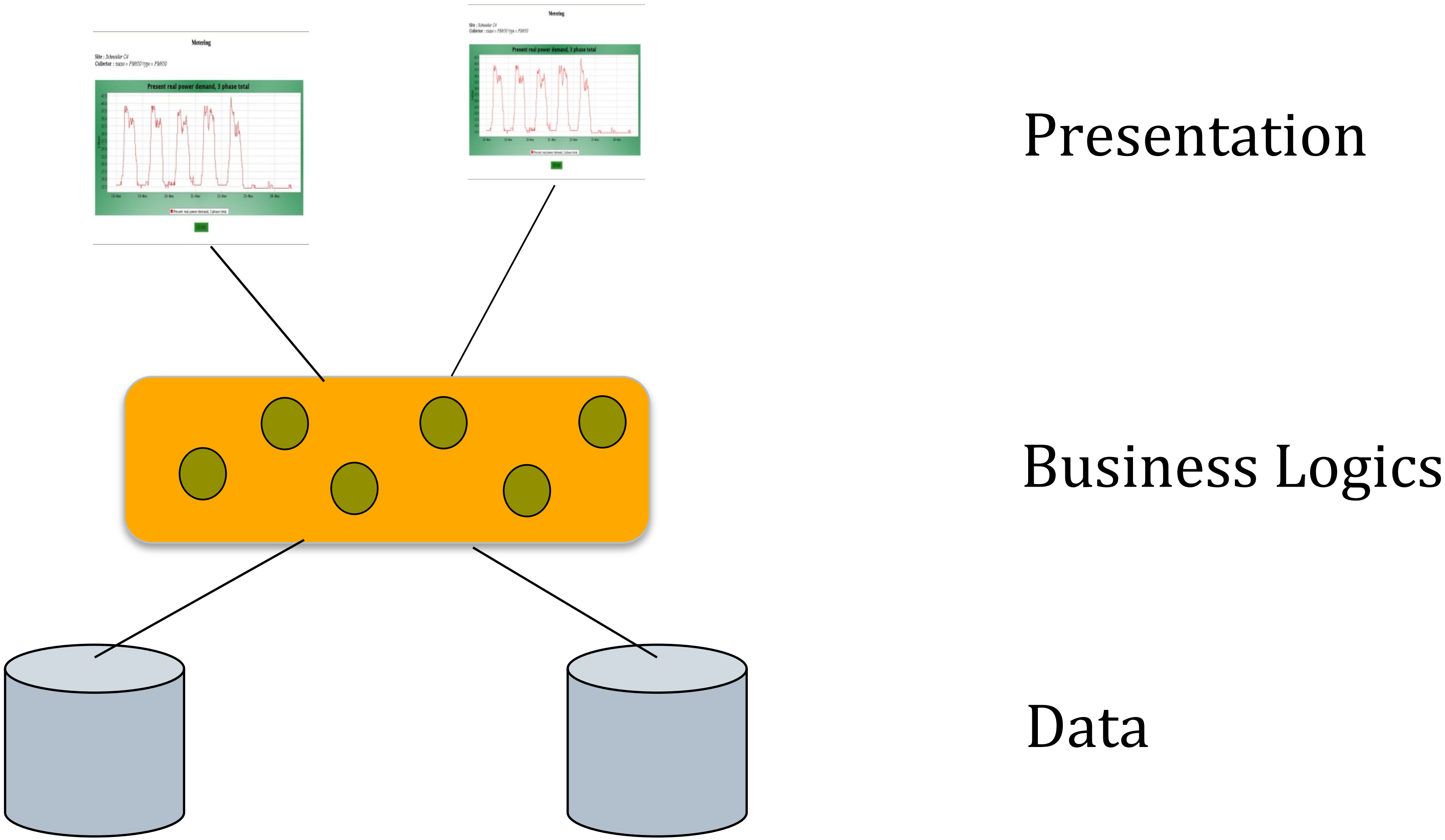
Limits

- Weak performance
  - Calculations must go through all the layers
  - Possible redundancies on the layers
- Error handling
  - Errors propagate on several levels
  - Hard to trace and debug
- Design is complex
  - Identify layers corresponding to appropriate abstractions

Example: communication stacks

Application	Diverse protocols
Presentation	Information structuration and semantics
Session	Dialogue control and synchronization
Transport	Creation of paquets and delivery guarantee
Network	Selection of a path
Data binding	Detection/correction of errors in bits sequences
Physical	Bits transmission

Example: information system



Conclusion

- Some persons see it as the “best” type of architecture ...

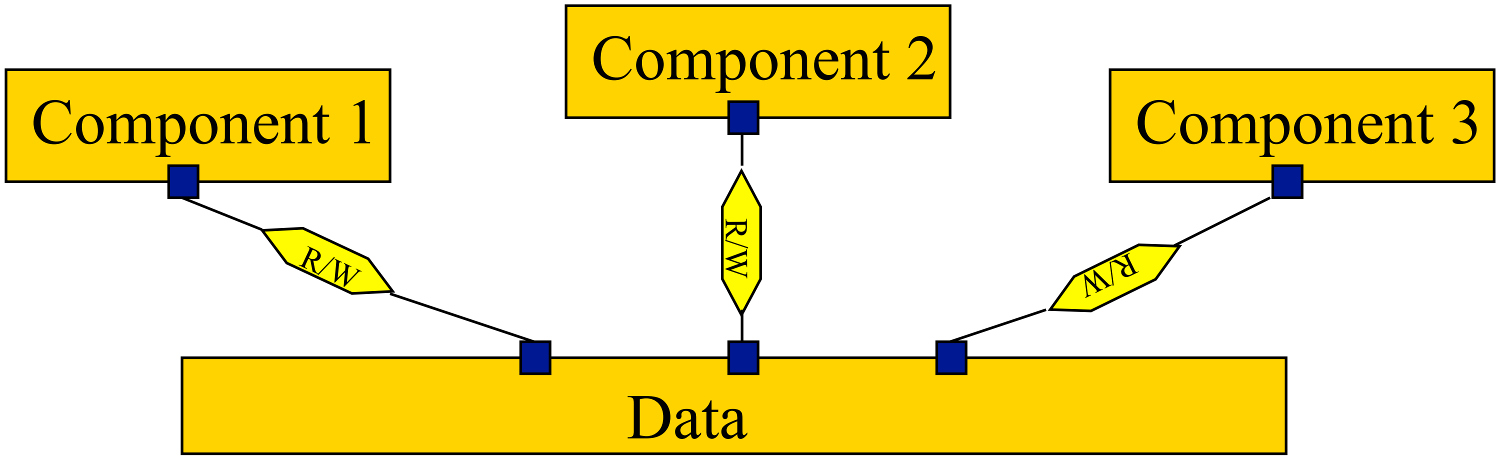
- Use it when
  - Levels of abstraction can be identified

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style and blackboard style
- Conclusion

Shared memory style

- Structure the system into components communicating solely through a shared database
  - Synchronous or asynchronous interactions, initiated by the emitter
  - Interactions are deterministic and non continuous



Style characterization

Elements

- Computing components
- Shared memory (a component too)
- Ports: write and read interfaces
- Connectors: no specification

Computing model

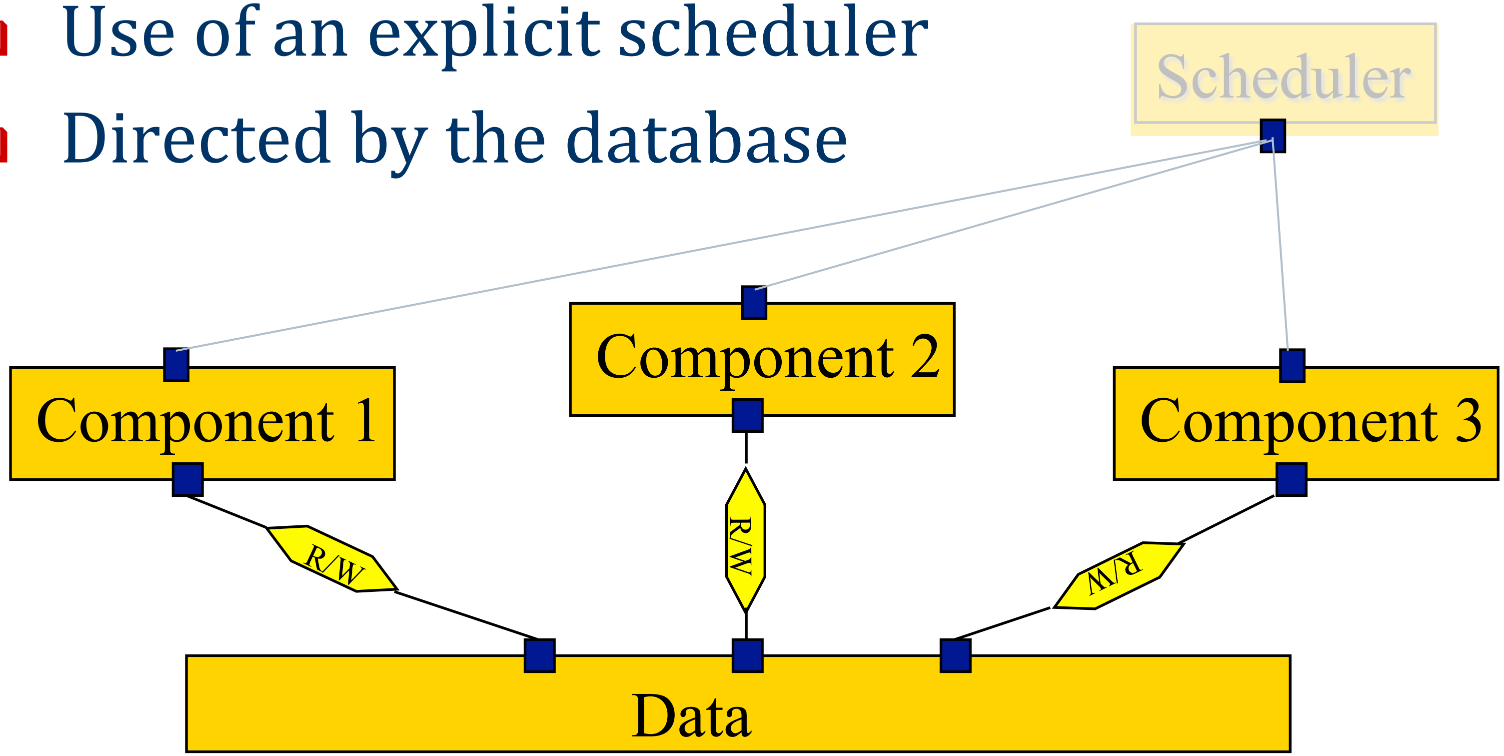
- Communication is made through the shared memory. Components do not know each others.

Constraints

Topology: components are all connected to the shared memory, and only it.

Implementation

- Key point: activation of components
  - Use of OS task scheduling
  - Use of an explicit scheduler
  - Directed by the database



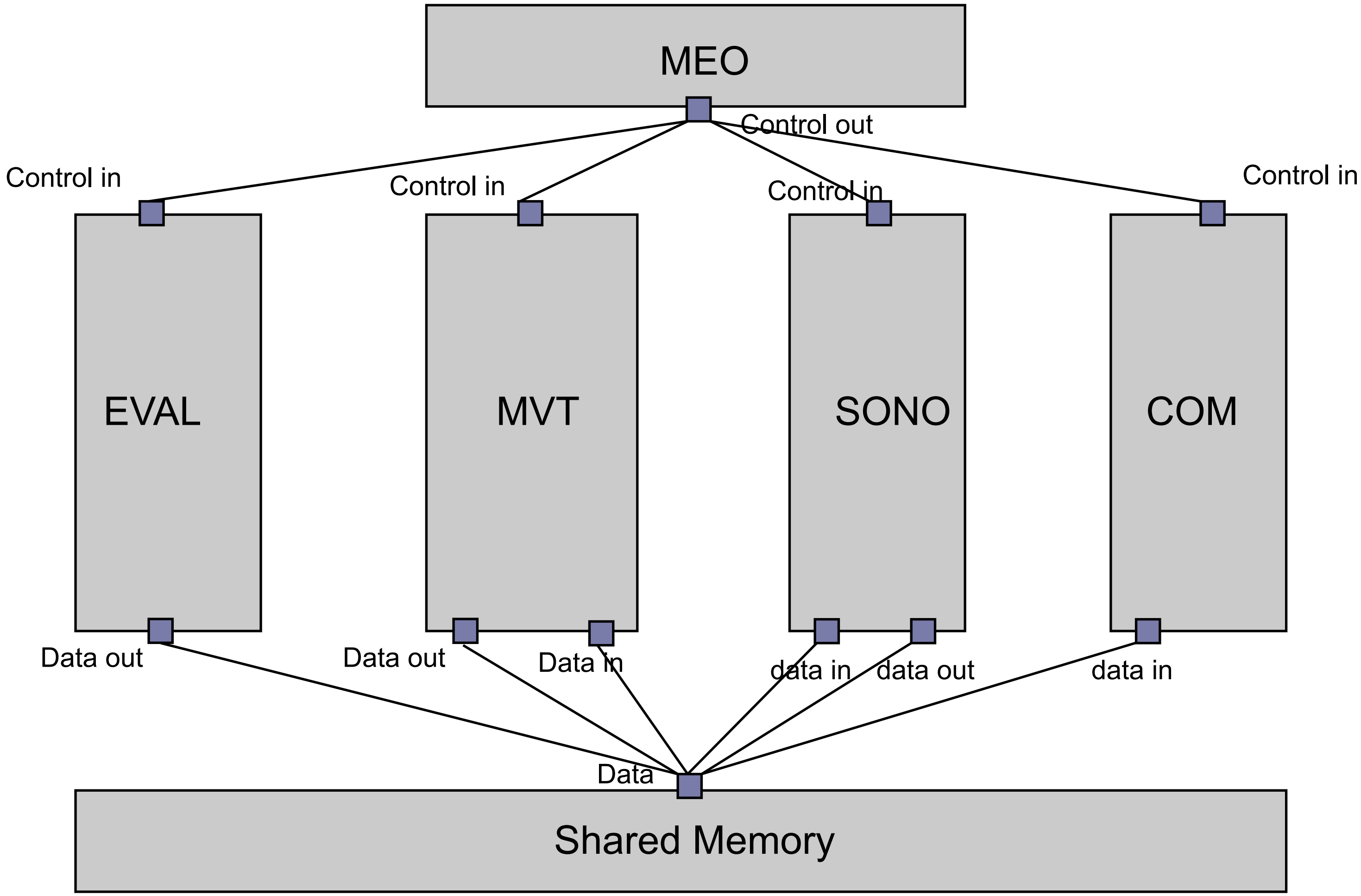
Advantages

- Performance
  - No communication cost
  - Indirection has a very low cost
  - Be careful not to have a bottleneck
- Robustness
  - Restart is often possible if the shared memory in not damaged
  - Shared memory is however a sensitive point

Limits

- Unclear
  - Communications are not explicit
  - The state of the shared memory is hard to interpret
- Security
  - The shared memory is a key element
- Maintenance
  - Modifying the shared memory is very costly
  - Side effects may be hard to identify

Example: simulation



Conclusion

- Widespread in embedded systems …

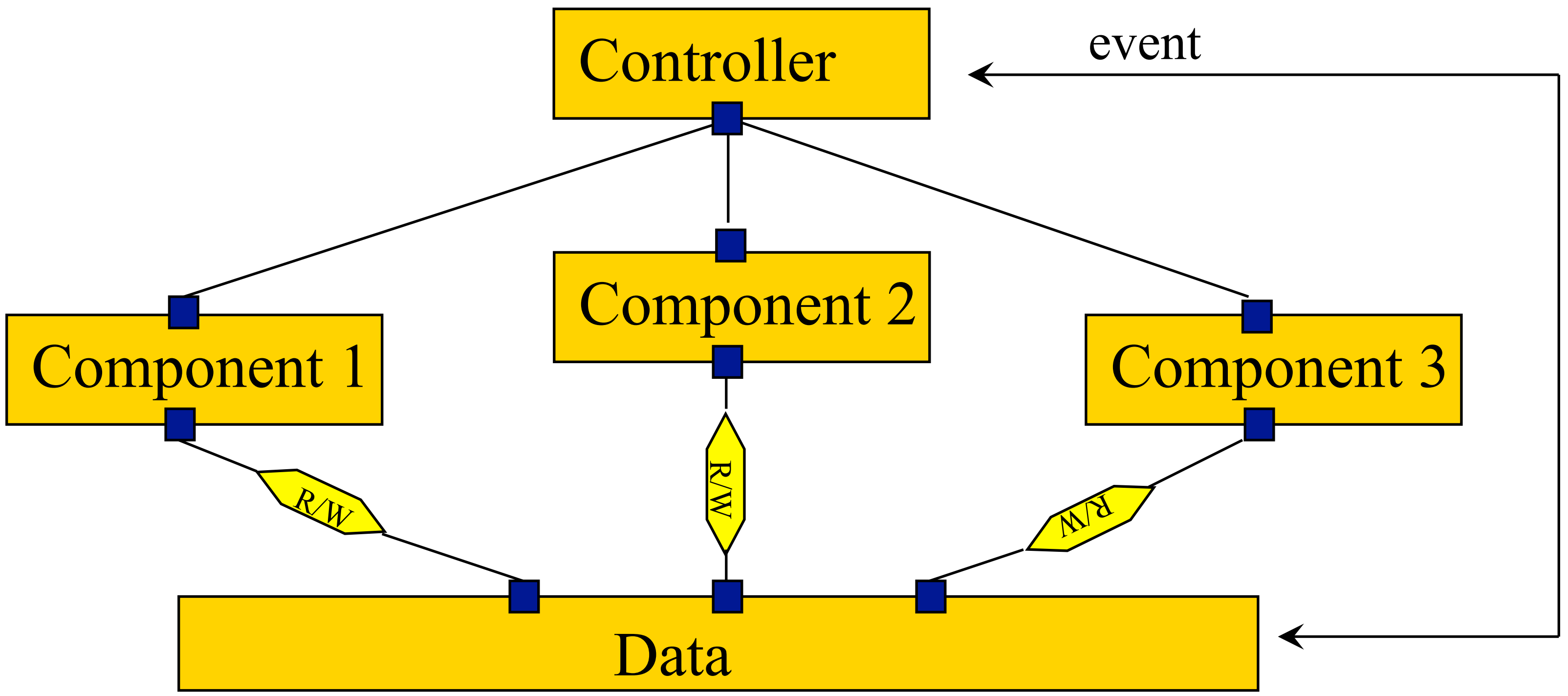
- Use it when
  - Performance!

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style and blackboard style
- Conclusion

Blackboard style

- Structure the system into components communicating solely through a shared database and opportunistically activated



Style characterization

Elements

- Knowledge sources
- Blackboard
- Ports: write and read interfaces on KS
- Ports: event management on Bb and Controller

Computing model

- Communication is made through the shared memory. Components do not know each others.
- Components are activated by the controller
- Opportunistic control

Constraints

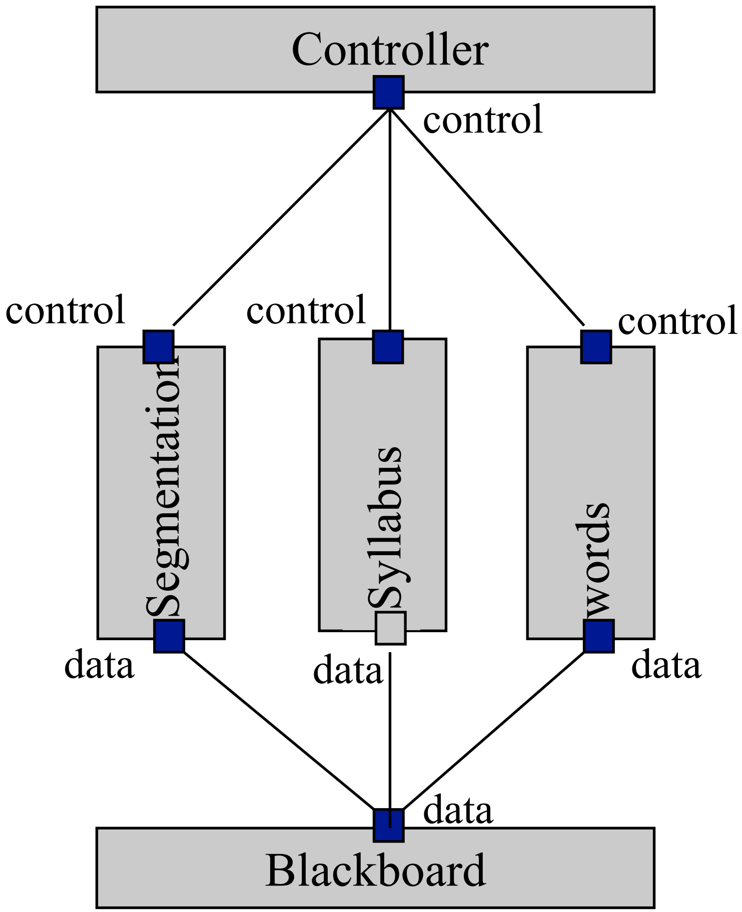
Topology is constrained:

- KS are all connected to the shared memory
- Controller is connected to KS and blackboard.

Advantages and limits

- Advantages
  - Very well suited to exploratory domains
  - Adaptative
  - Robust and fault tolerant
- Limits
  - Hard to test and debug
  - Not efficient
  - Optimal solution is not guaranteed
  - Control strategy is very complex to define

Example


Controller
control
control
control
control
Segmentation
Syllabus
words
data
data
data
data
Blackboard

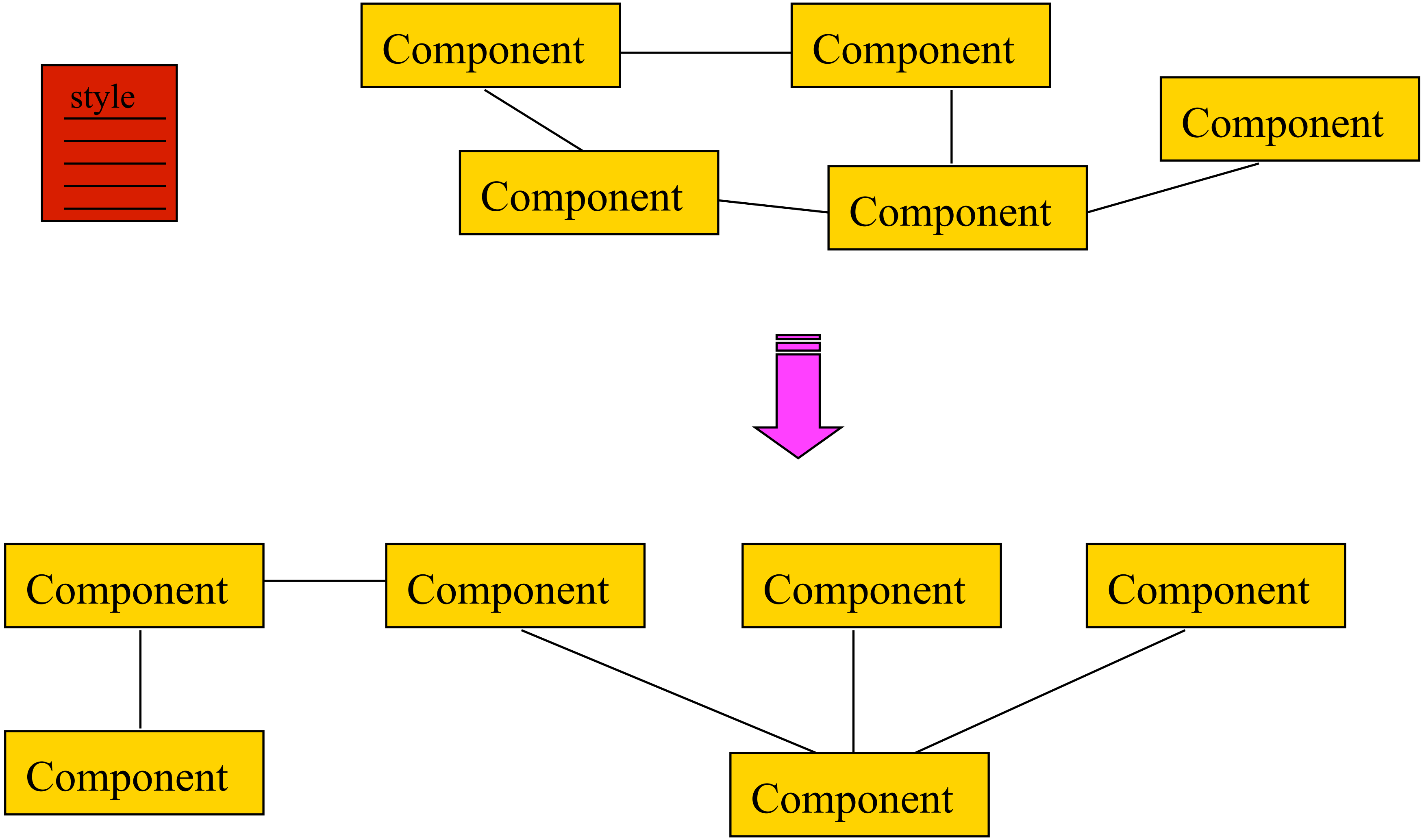
Example

- Cycab robot

Outline

- Definition
- Communication styles
  - Client / server style
  - Message-oriented style
  - Publish / subscribe style
  - Pipe and filter style
- Organization styles
  - Layered style
  - Shared memory style and blackboard style
- Conclusion

Goal



style
Component
Component
Component
Component
Component
Component
Component
Component
Component
Component
Component

Styles interactions

- An easily identifiable architecture is needed
  - Easier to understand, communicate, evaluate
- Many styles are used in a single system
  - Architect's role to find the right mixing