Tampereen yliopisto
Tampere University

Large Scale Software Design
Architectural styles

Hannu-Matti Järvinen, David Hästbacka

Architectural styles

- Partitioning architectural styles
  - Layer / tier architectures (partition by structure)
  - Pipes and filters architecture (partition by functionality)
- Service-based architectural styles
  - Client-server architectures
  - Peer-to-peer
  - Message-passing architectures (publish-subscribe)
- Special architectures
  - MVC architectures
  - Interpreter architectures

What are architectural styles?

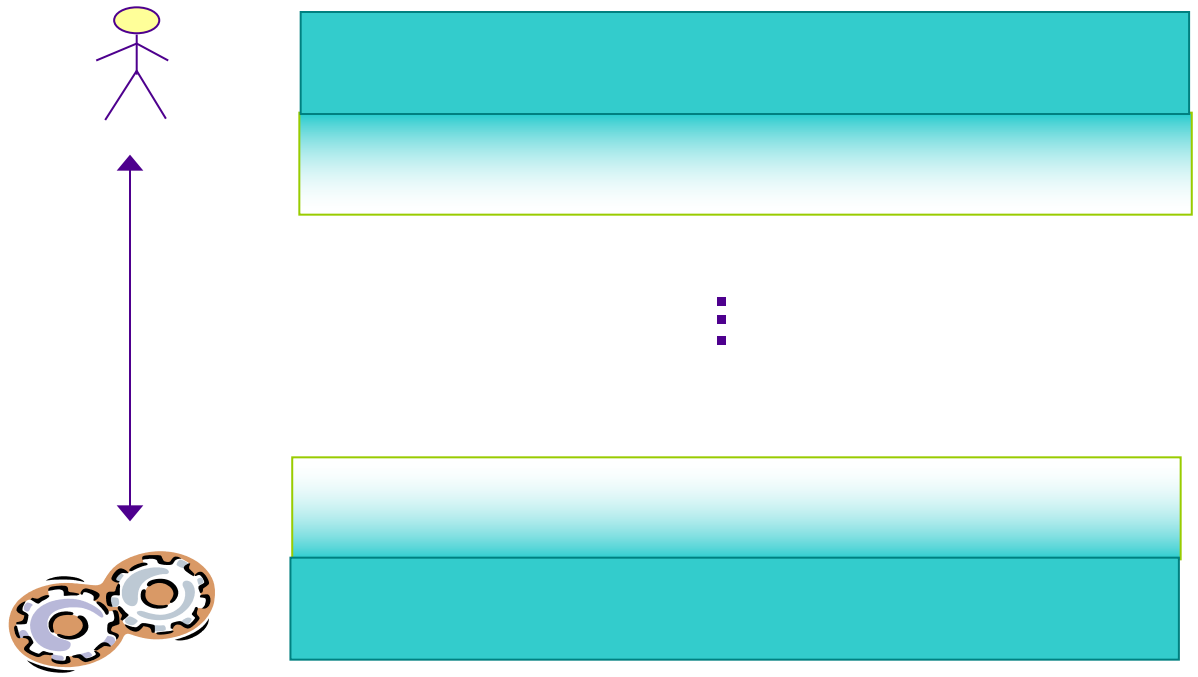
- Architectural style = a common model telling how the system is organised on the highest abstraction layer. Defines the general technical nature of the system.
- Architectural styles do not exclude each other!
- Question leading to a selection of an architectural style:
  - Does the system consist of components on different conceptual levels?
  - Is the process information the main purpose of the system?
  - Does the system share information or resources with individual applications?
  - Does the system consist of a set of components communicating with each other in ways not know in advance? Do the components change often?

Architectural styles and architectural patterns

- Pretty much the same thing (sometimes considered synonymous), but a little bit different point of view.
- An *architectural style* is a conceptual way of how the system will be created / will work.
- An *architectural pattern* describes a solution for implementing a style at the level of subsystems or modules and their relationships.

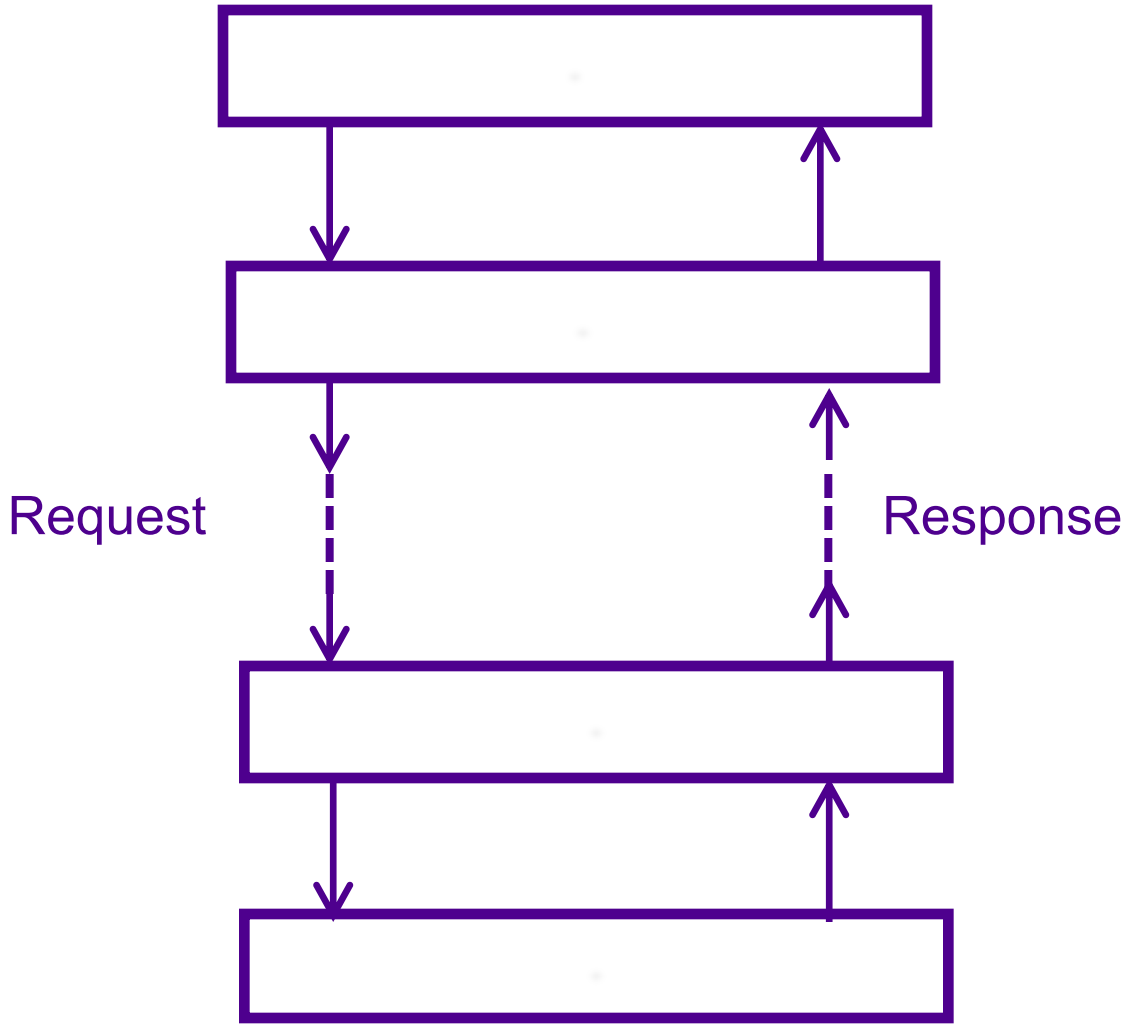
Layer / tier architecture 1

- System is organised as layers with increasing abstraction levels.

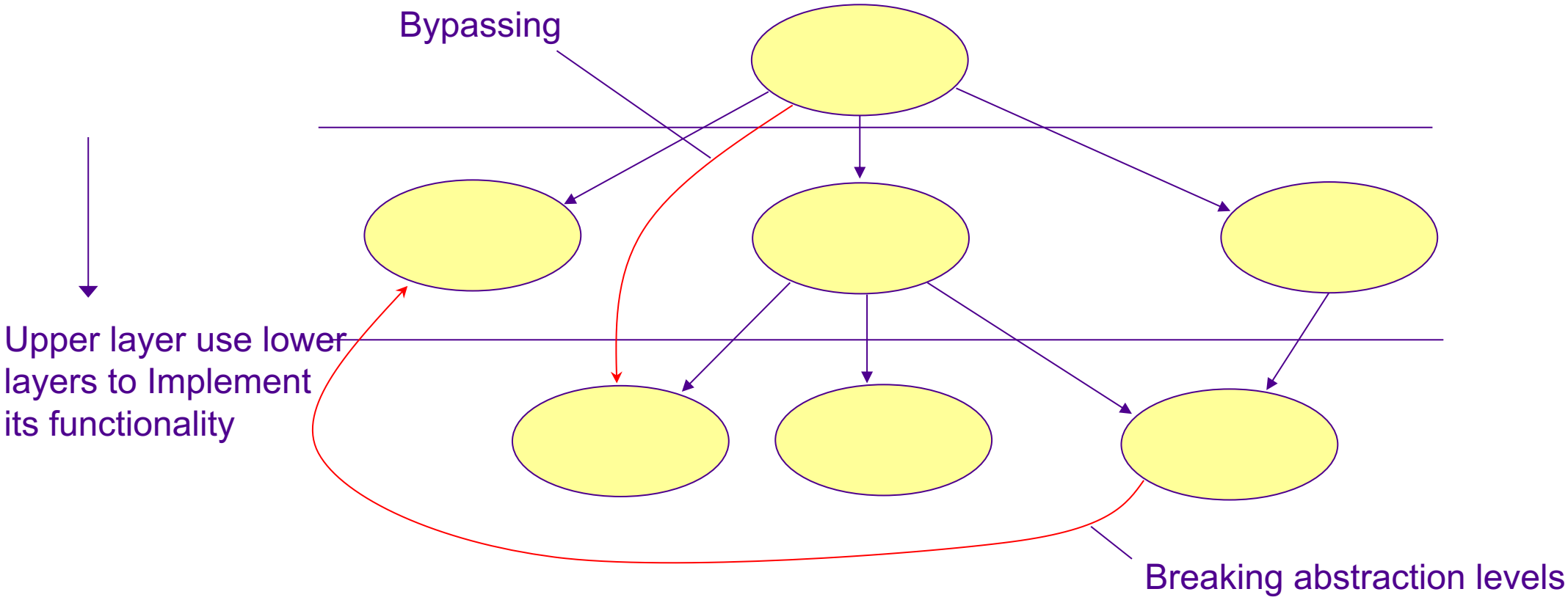


Tier Architecture 2

- Hierarchical layers
  - Provide services to the upper layers (e.g., hide the heterogeneity)
  - Use lower layers' services
- In pure layer architectures, service requests progress from top to down

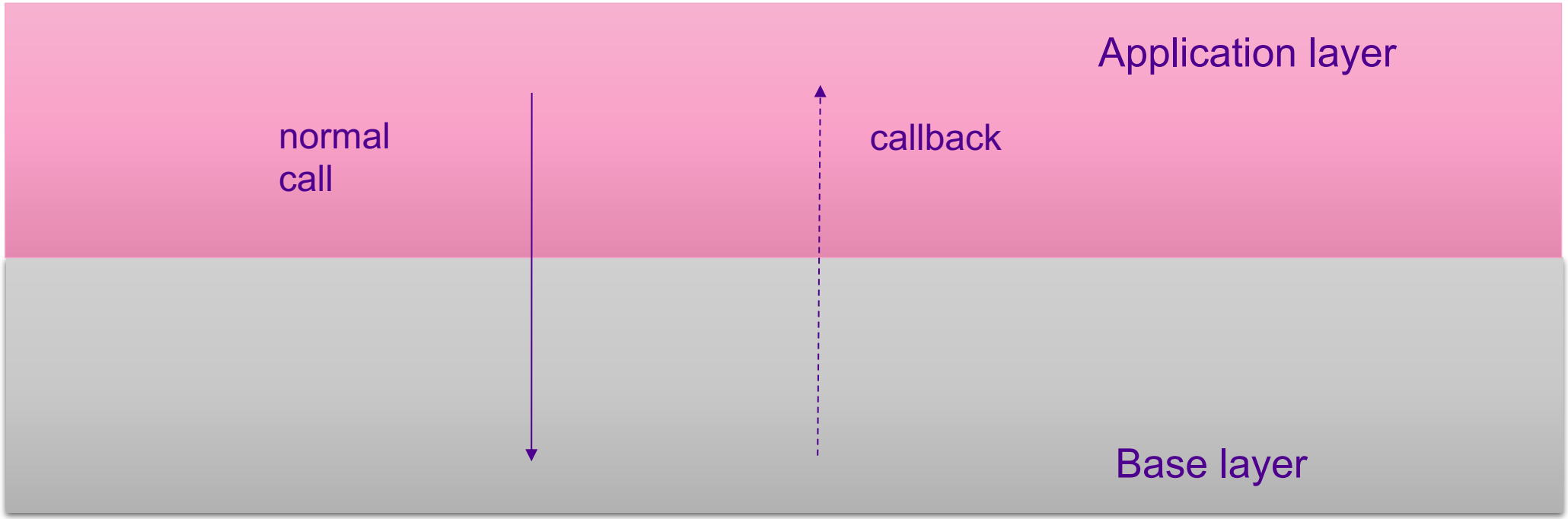


Breaking and bypassing abstraction layers

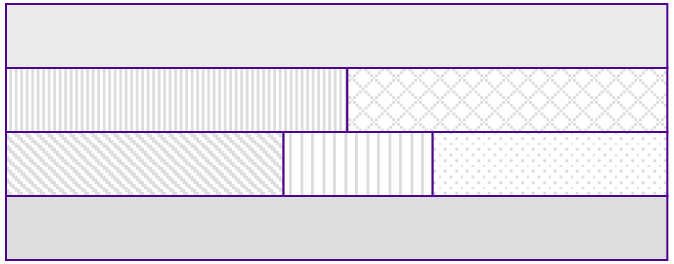


Callbacks in tier architecture

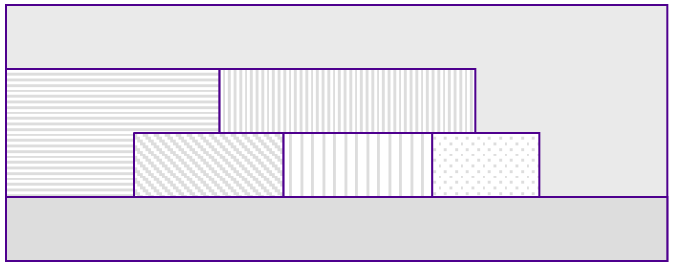
- Callbacks can be used to break the idea of tier architecture without making lower level dependent of the upper.



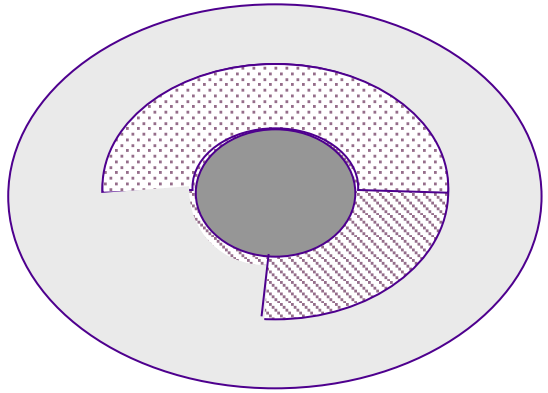
Describing tier architectures: hamburger



No bypassing

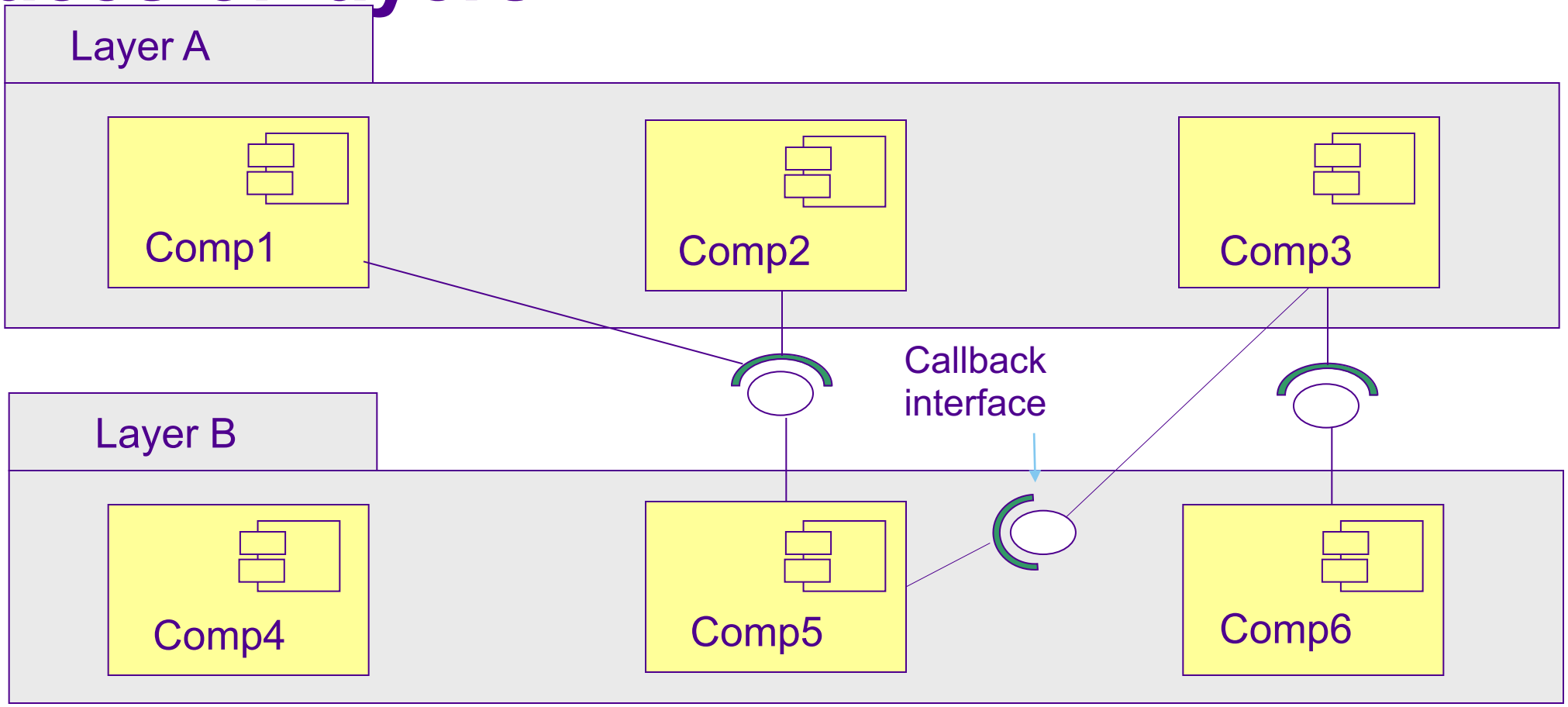


Bypassings



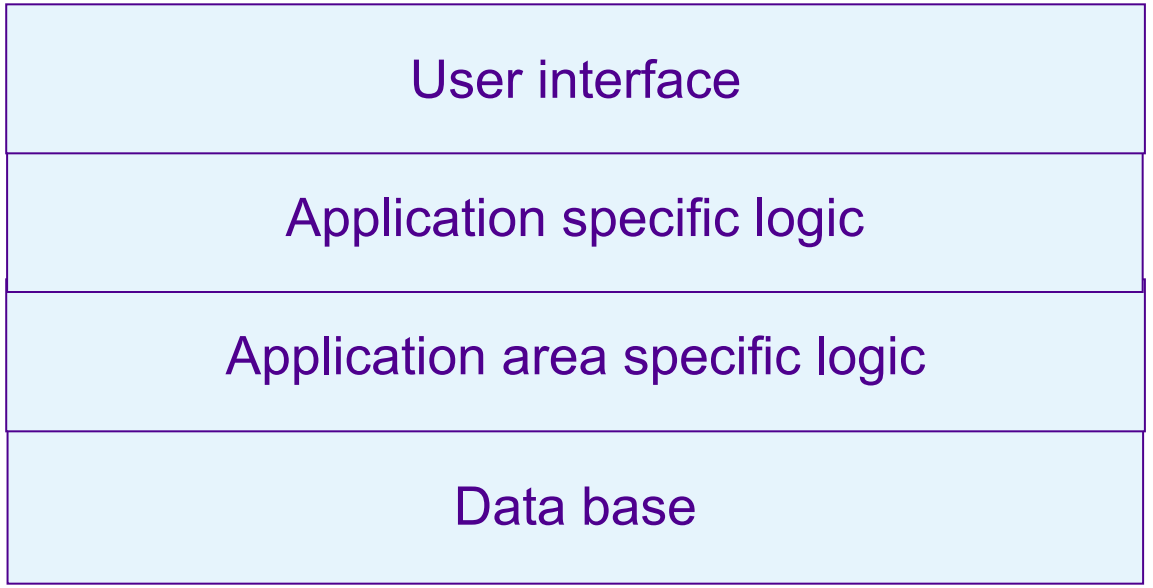
Bypassings

Interfaces of layers

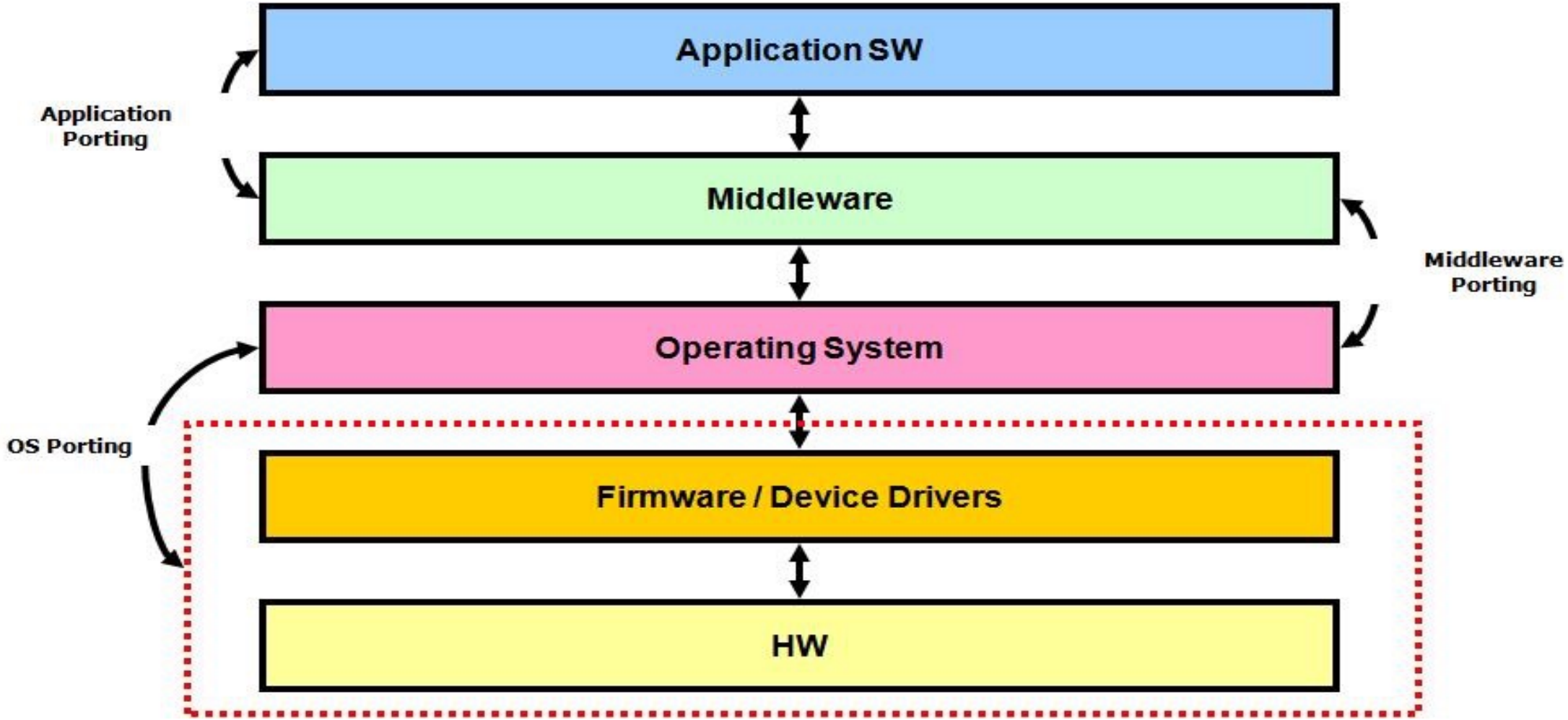


Note: Facade

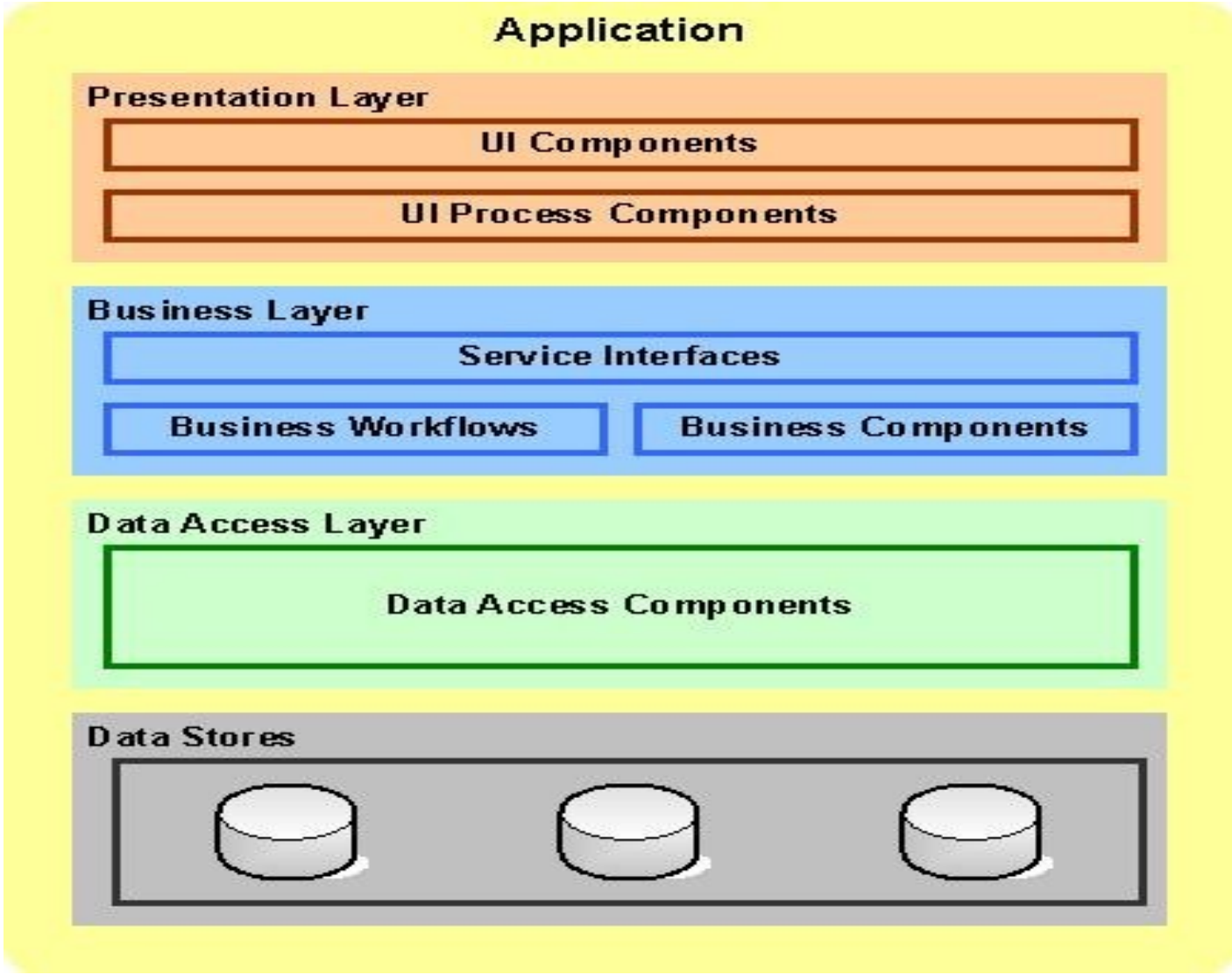
Example: typical business system



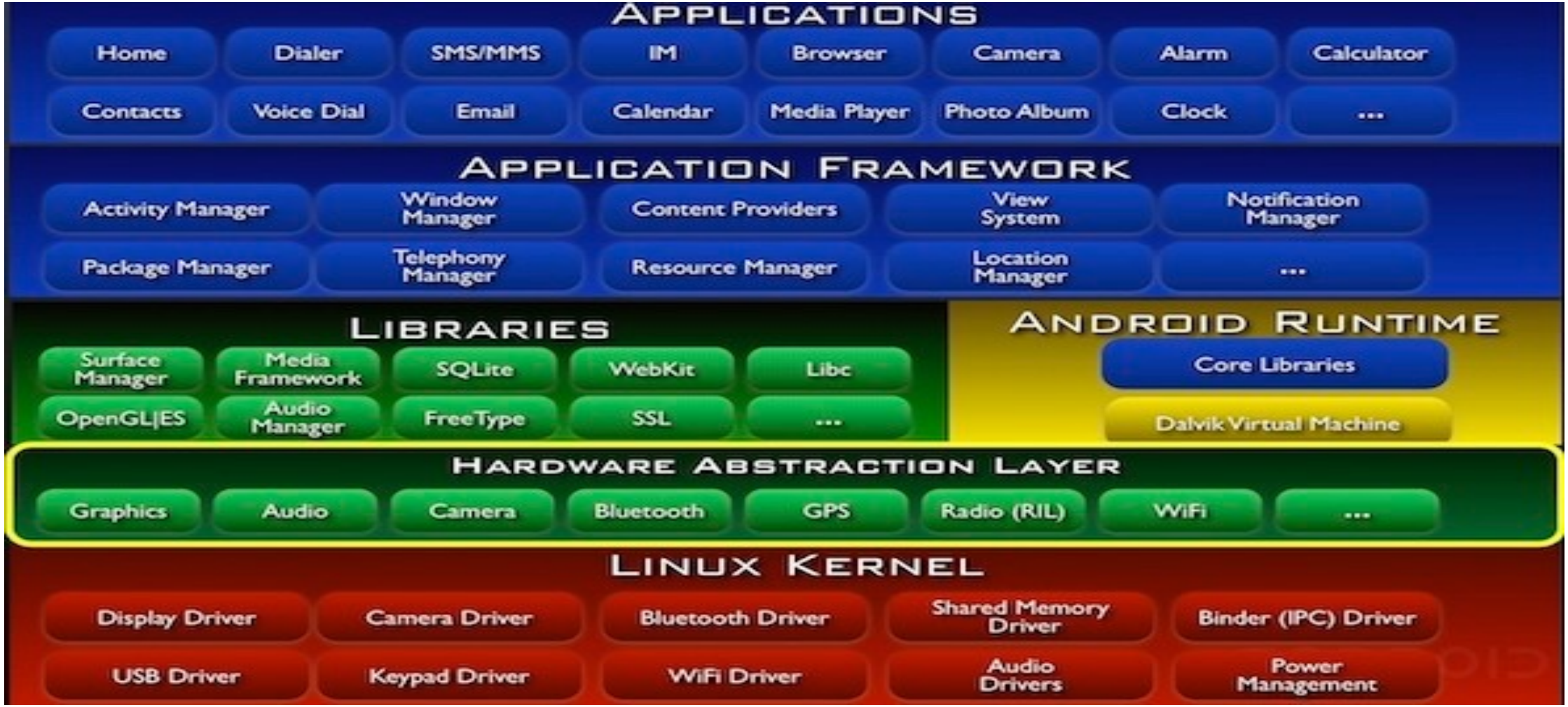
Example cont.



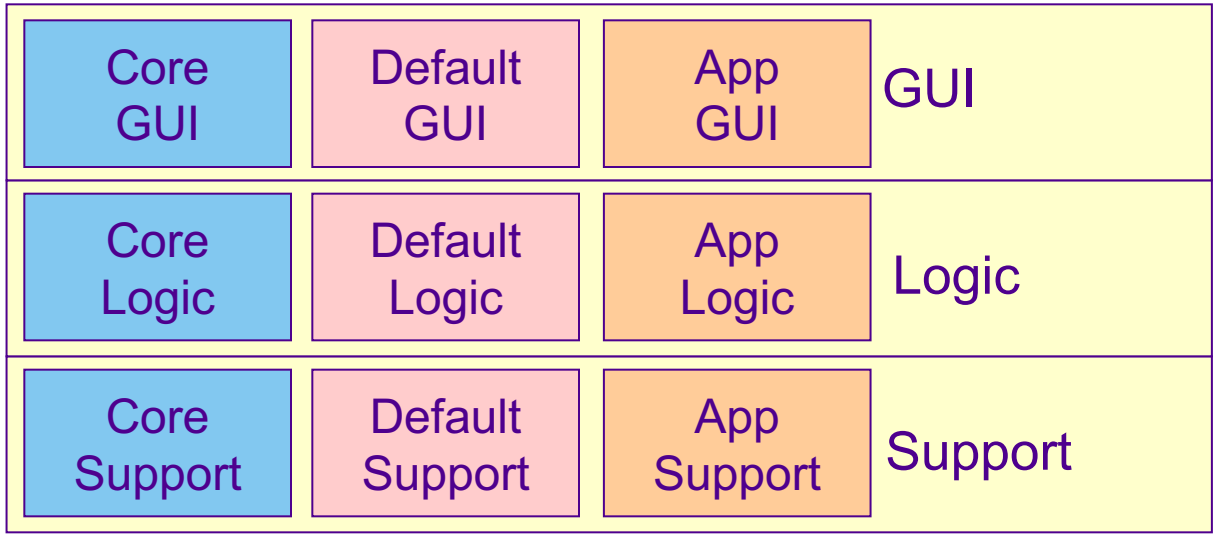
Example cont.

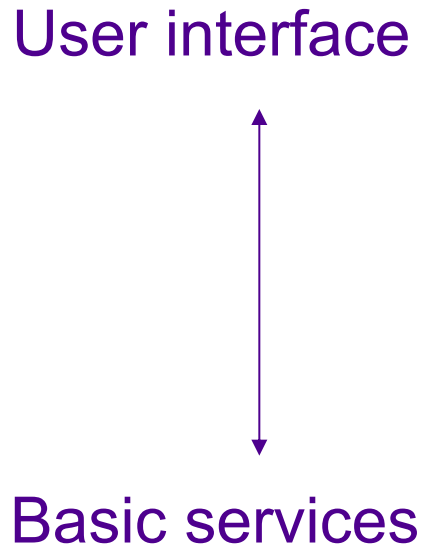


Layers and components

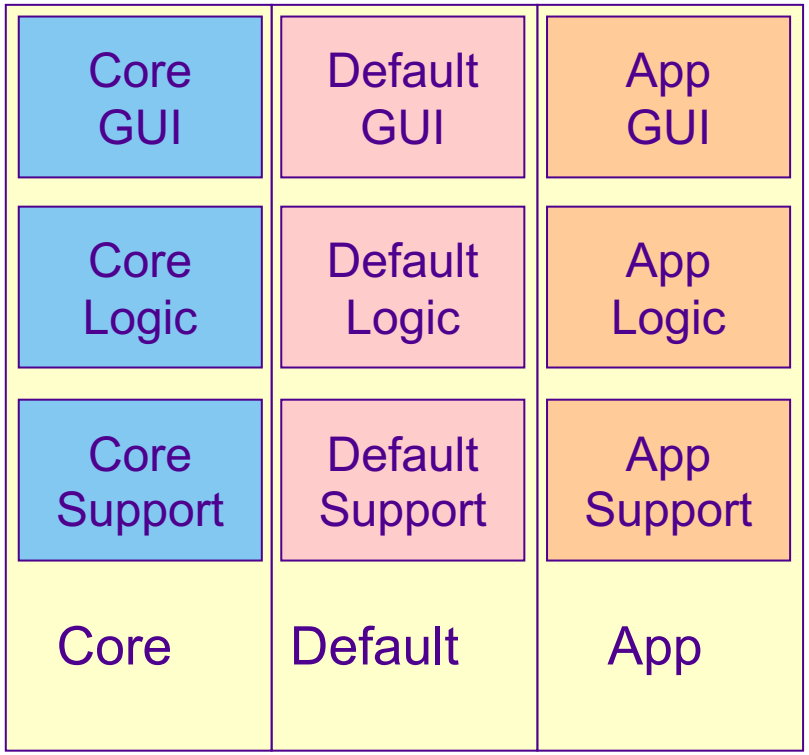


Tier architecture can be multidimensional (vertical)





Tier architecture can be multidimensional (horizontal)



General purpose — Product specific

Domain – Company - Product

Dependency analysis of layers 1

Row layer depends on layer in column	P1	P2	P3	P4	P5
P1		X	X		
P2			X	X	
P3				X	X
P4					X
P5					

OK,
Bypass ratio
= 1

Dependency analysis of layers 2

row depends on col	P1	P2	P3	P4	P5
P1		X	X		
P2				X	
P3		X		X	X
P4					X
P5					

Not OK,
What can be done?

Change the positions of P2
and P3, if P2 is not clearly on higher abstraction level.

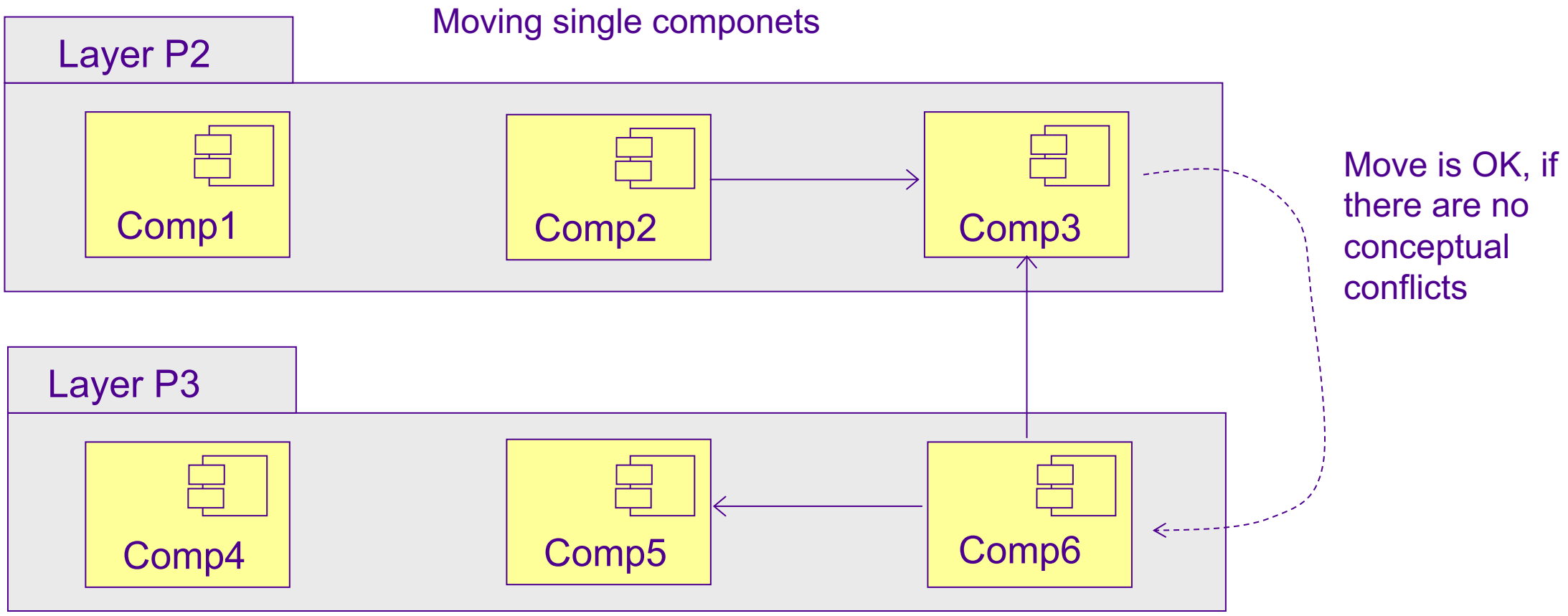
Dependency analysis of layers 3

row depends on col	P1	P2	P3	P4	P5
P1		X	X		
P2			X	X	
P3		X		X	X
P4					X
P5					

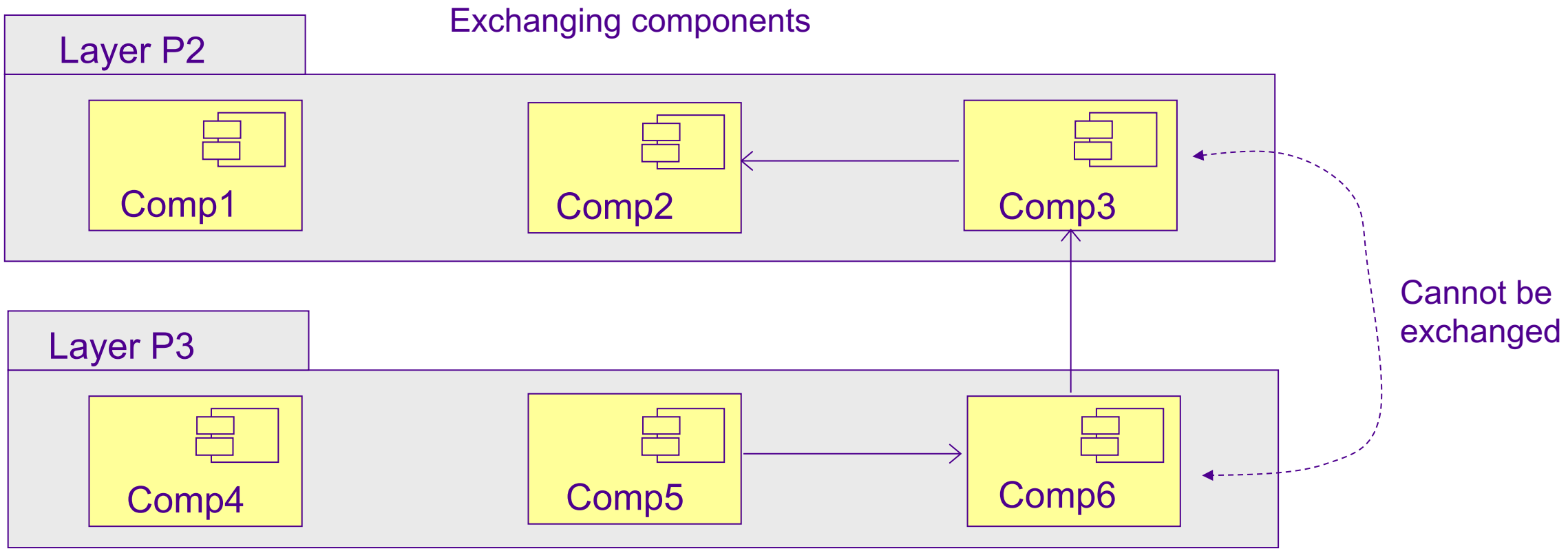
Not OK
What can be done?

Combine P2 and P3?

Dependency analysis of layers 4

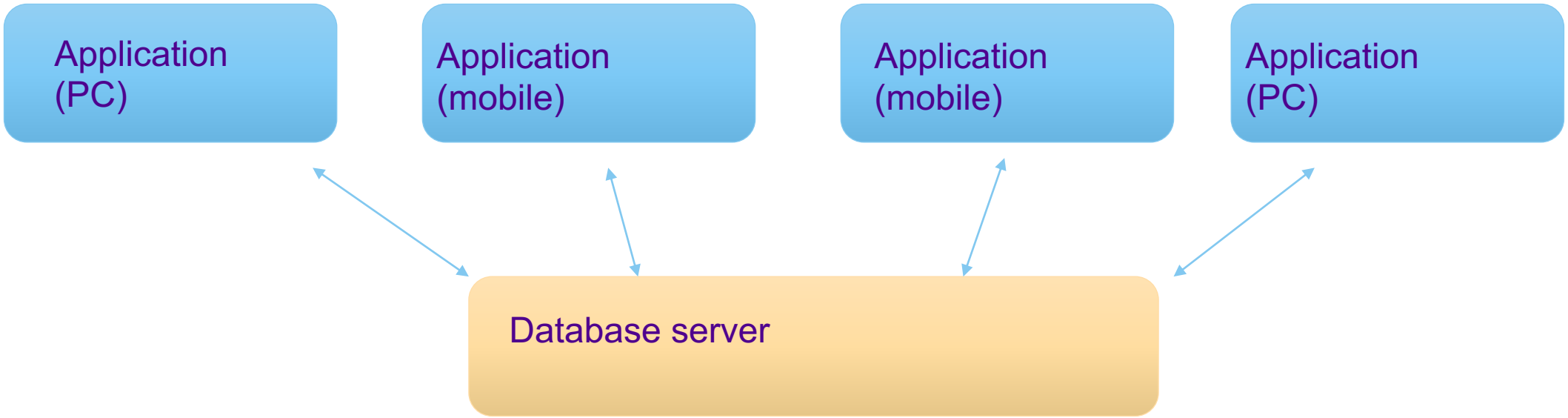


Dependency analysis of layers 5



Distribution of tier architecture

- Tier architecture is easy to distribute
  - Communication between layers by messages
- Two-tier architecture distributed:

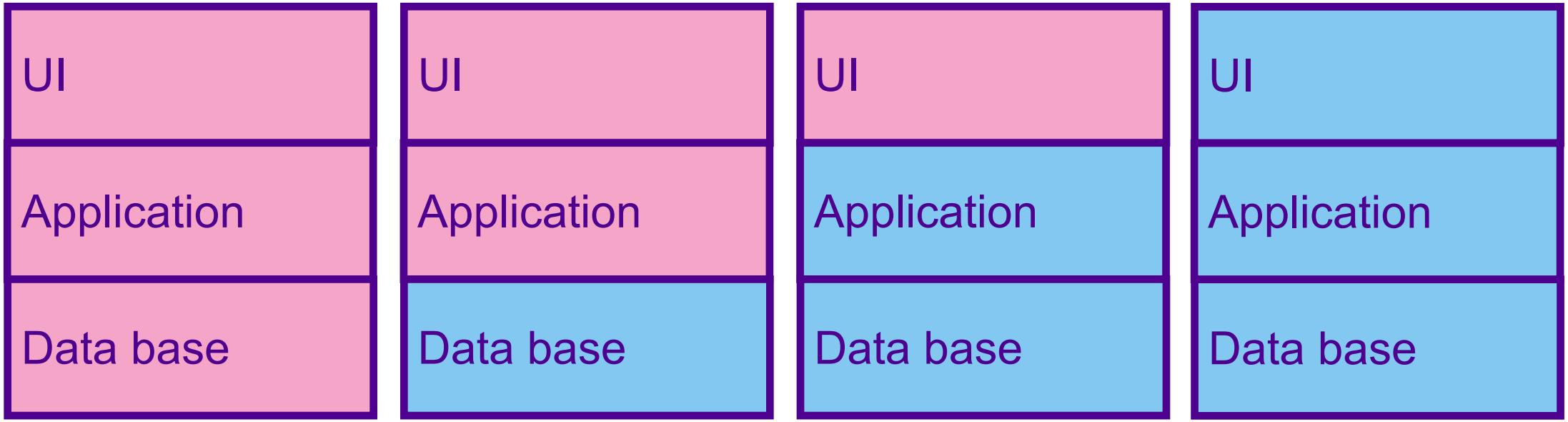


Three tier architecture

- Common in web:
  - Presentation layer / tier / front-end: part running in browser
  - Application / logic layer: application level functionality, code (Ruby, Java, PHP, ASP, .NET, Perl, ...) executed by application server.
  - Back end / Data tier: data base and management of data base, how to obtain stored data.

The right place to distribute is not self-evident

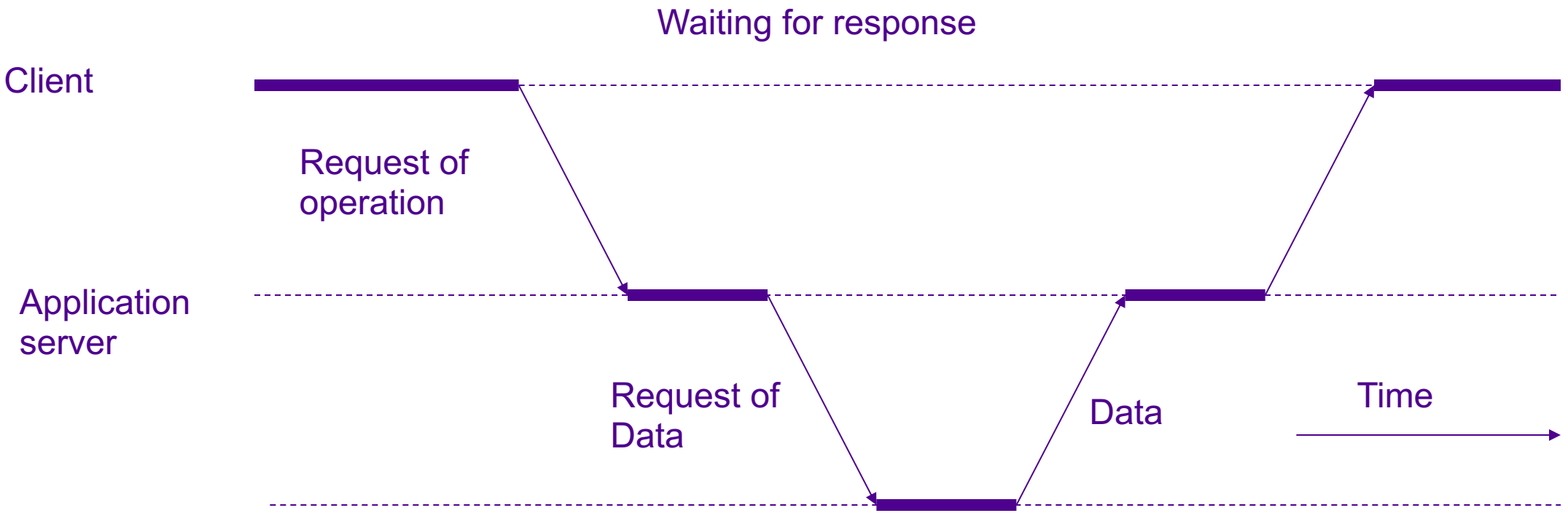
Client device



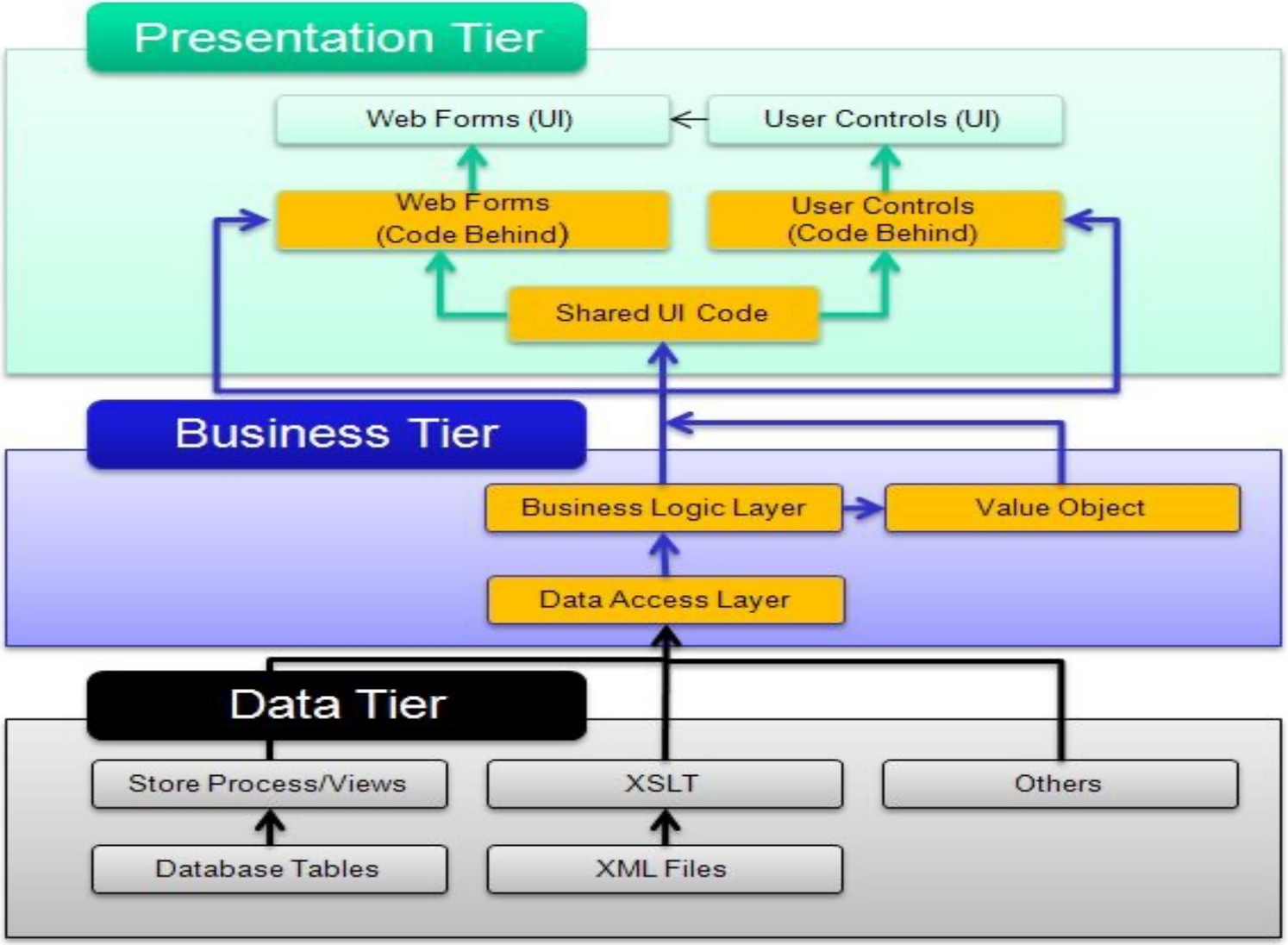
Server device

Request and Response

- Request and Response behaviour



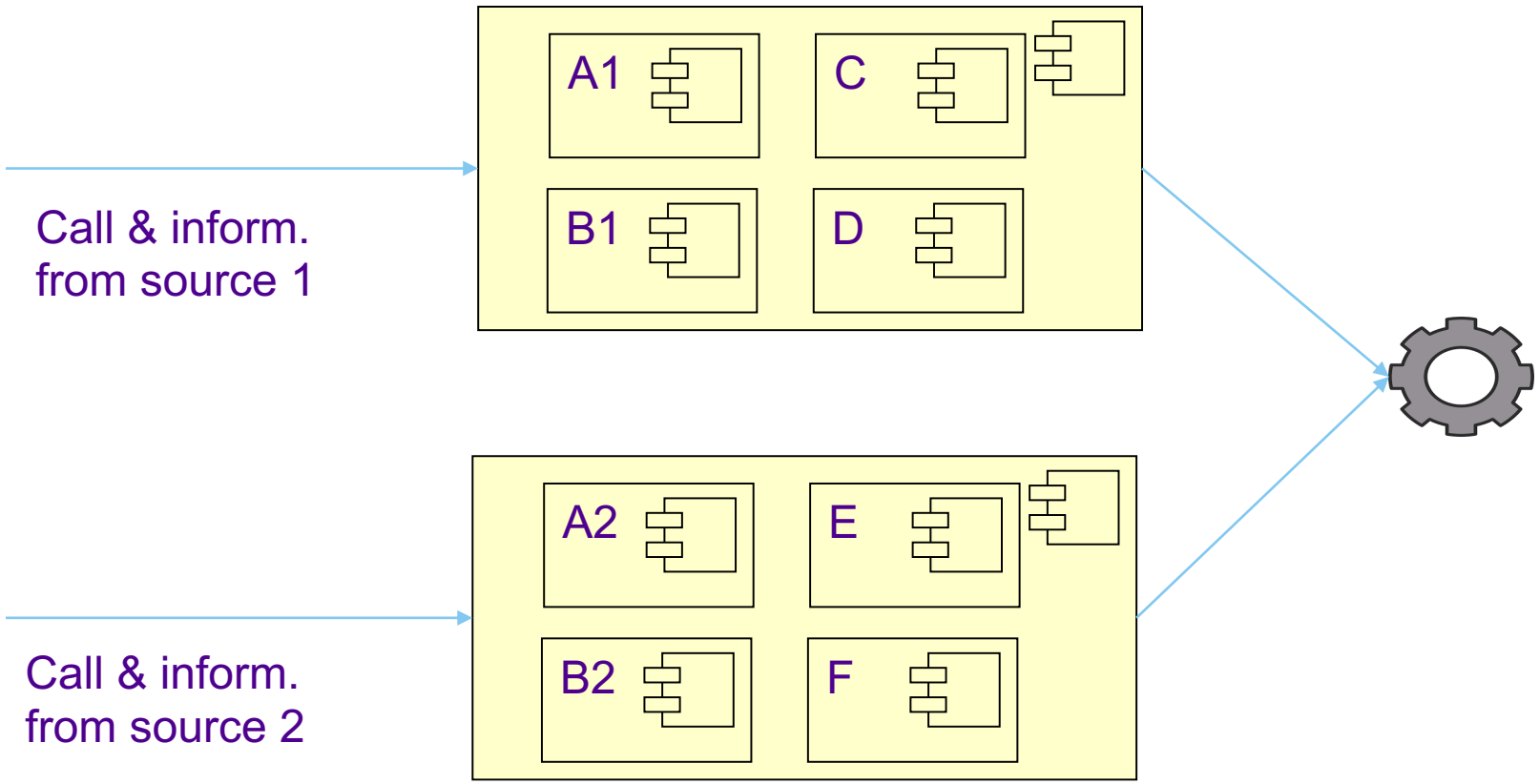
Example of three layers



Pros and cons of tier architectures

- Pros
  - Applicable in most cases
  - Support understanding and mastering of the system
  - Decrease dependencies (maintenance, adaptability)
  - Easy to connect with the organisation of the company (Conway)
  - Support reuse (product framework)
  - Distribute easily
- Cons
  - Performance can be compromised (indirection)
  - May lead to unnecessary or repeated computation
  - Exception handling

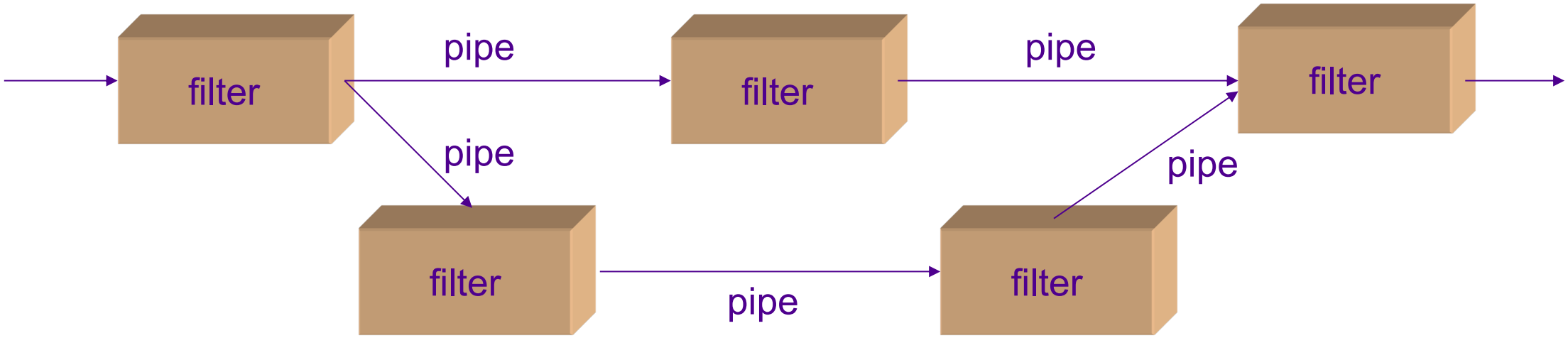
New problem area



- Separate, independently developed components.
- Both include some functionalities, but adapted to other internal components.
- No reuse.
- Optimisation of performance is difficult internally.
- Adaptation is difficult

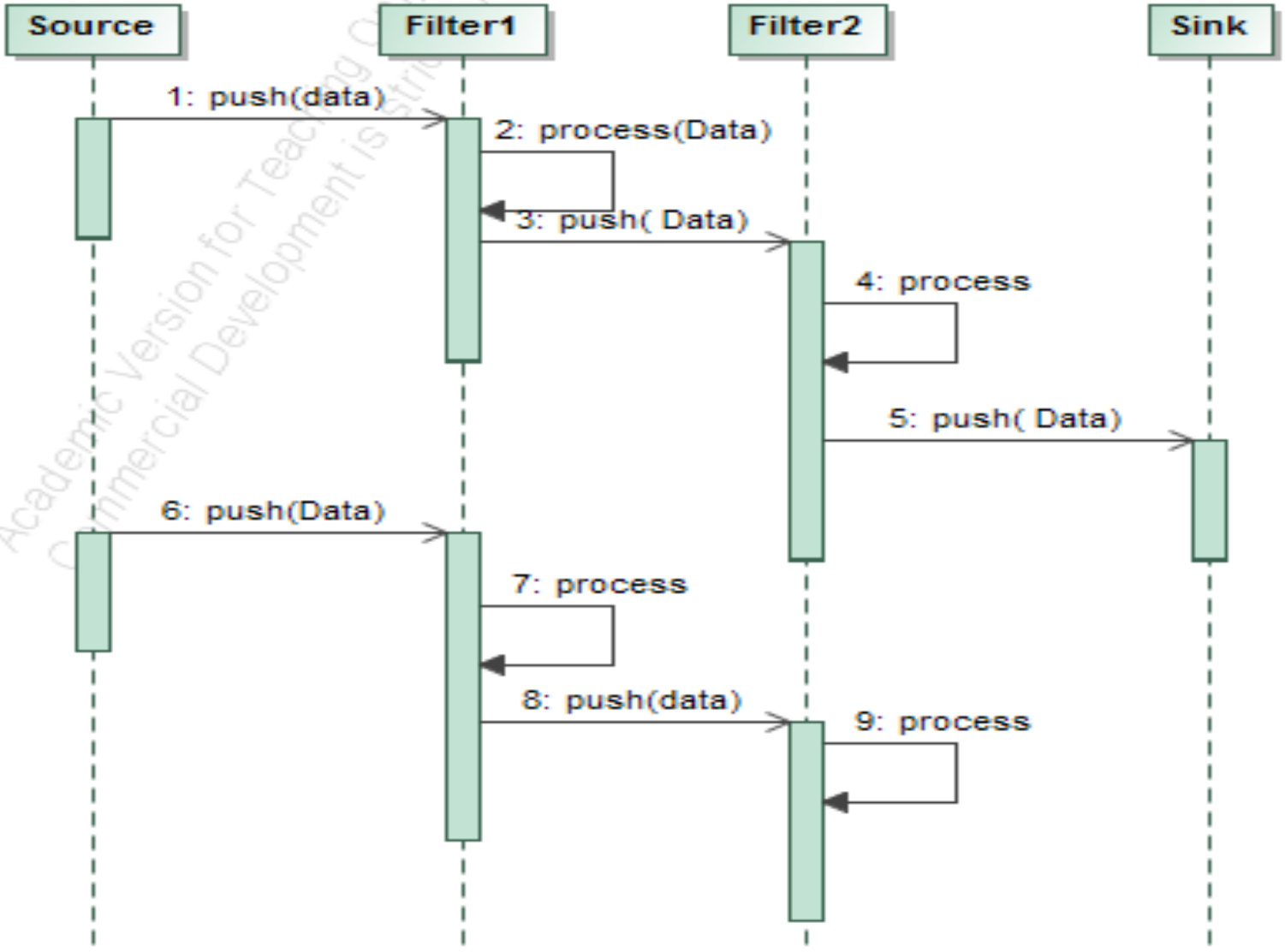
Pipes and filters architecture

- Pipes and filters architecture consists of components (filter) that produce and consume data items, and channels (pipes) that carry data items from component to component.

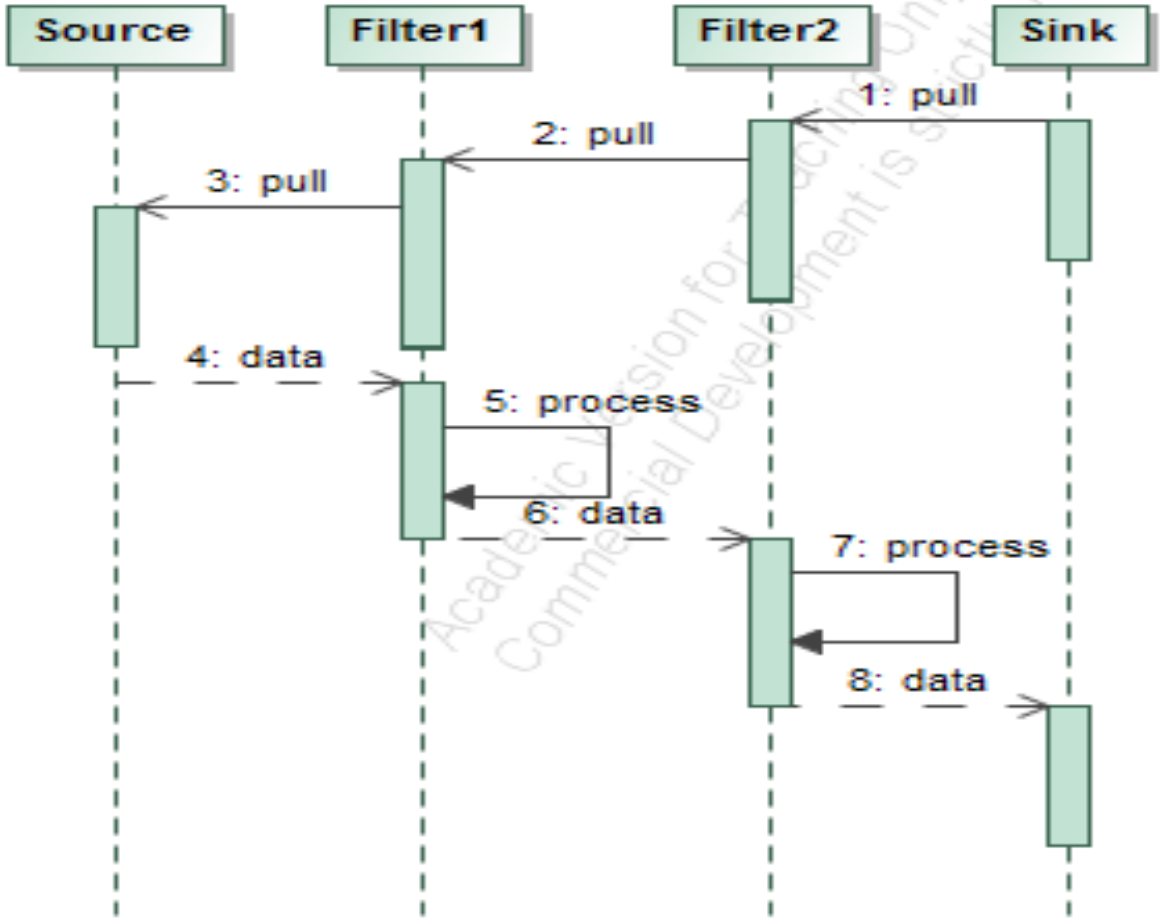


Special case: pipeline

Control implementation in pipelines: push type



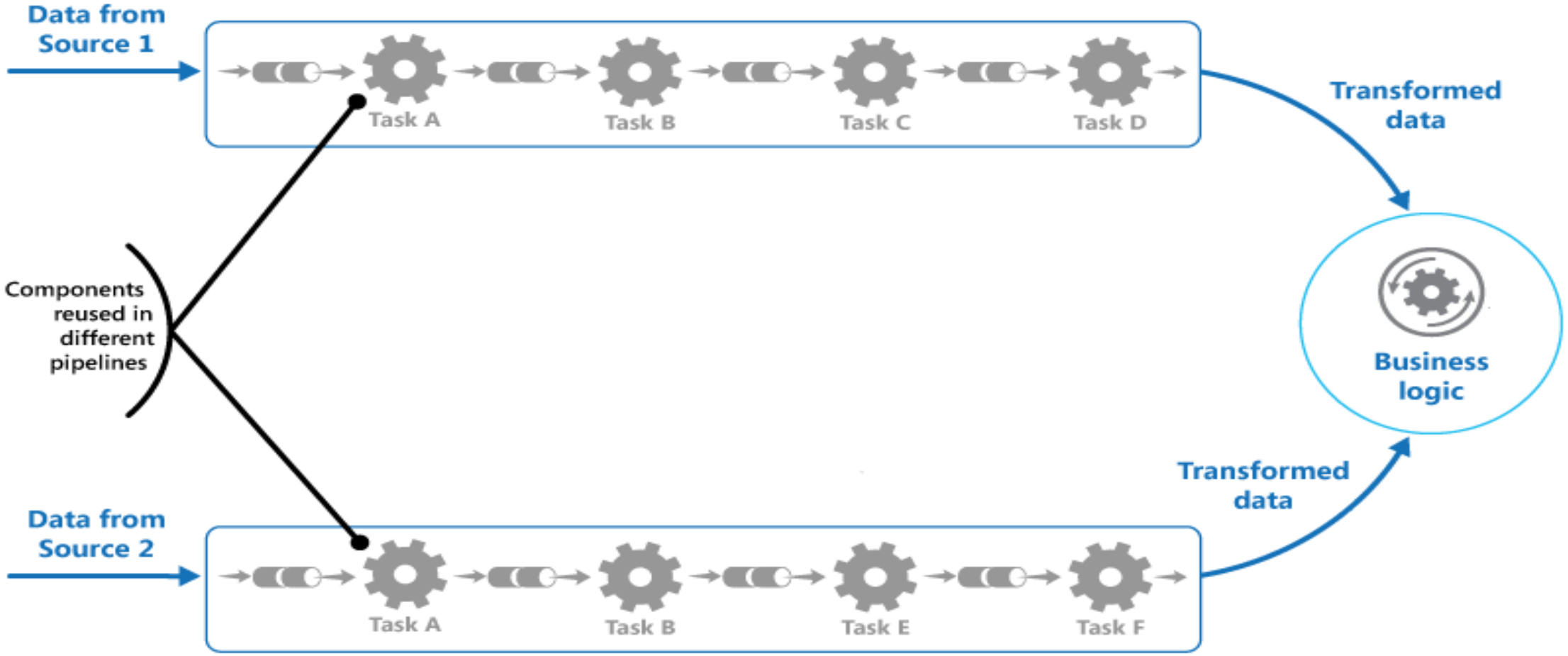
Control implementation in pipelines: pull type



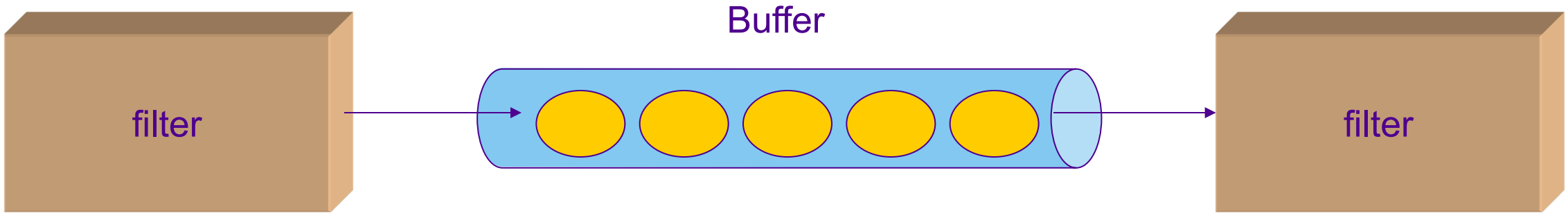
Typical points of pipes and filters architecture

- Processing units are working independently (they don’t share state information)
- Processing units do not know each other, only the data format required by the channels (pipes)
  - The format does not need to be the same for between all filters
- Information can be processed piecewise.
- Units are stateless.

Example of pipes and filters



Concurrent units: buffering



- Each unit in its own process
- Slowest unit defines the total time
- The sizes of buffers are critical
- The buffer is typically a queue structure

Cont.

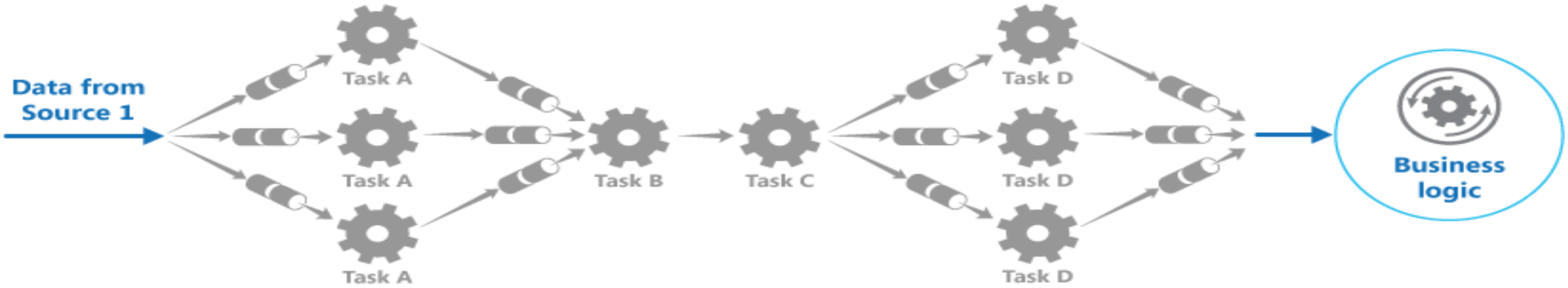
filter, retrieves a task from the queue

Incoming message

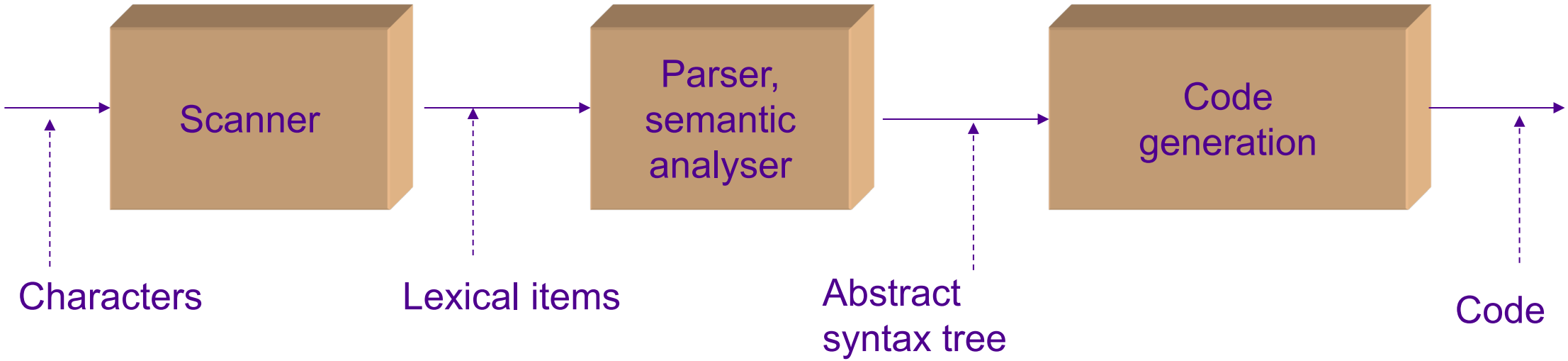
Message queue

Message fully processed

Load balancing

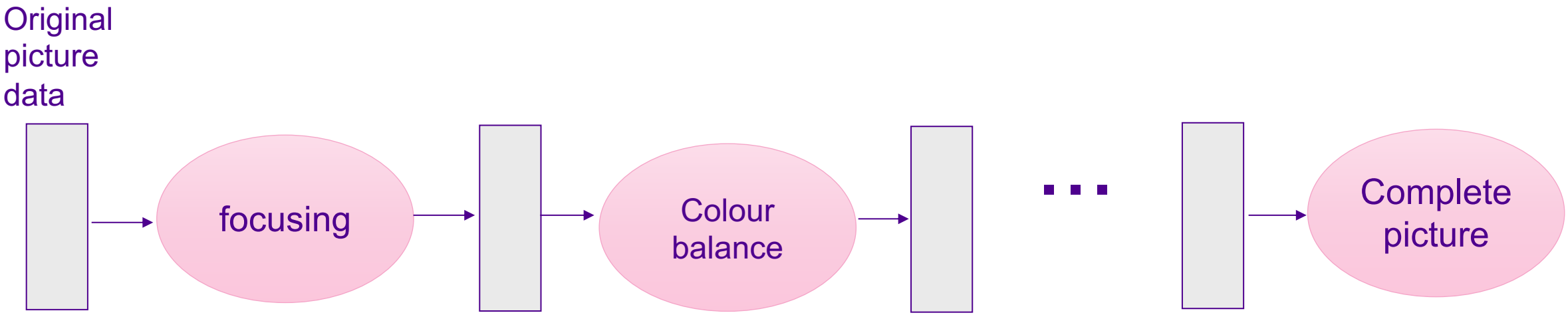


Example: one-pass compiler



- Problem: context-aware processing of source code
- Solution:
  - Data is gathered to a global symbol table
  - Parser may act as the main program

Example: photoshop lightroom



- The original picture is saved, the complete picture in the program is the original + operations
- A filter can be removed -> picture changes; exchange filters -> picture changes again
- Converting e.g. for net publishing or paper printing
- Ability to select the same operations for the whole picture library

Examples

- Command lines (Unix, Linux), and piping of commands.
- Graphics (GPU computation) and tasks that can be parallelised.

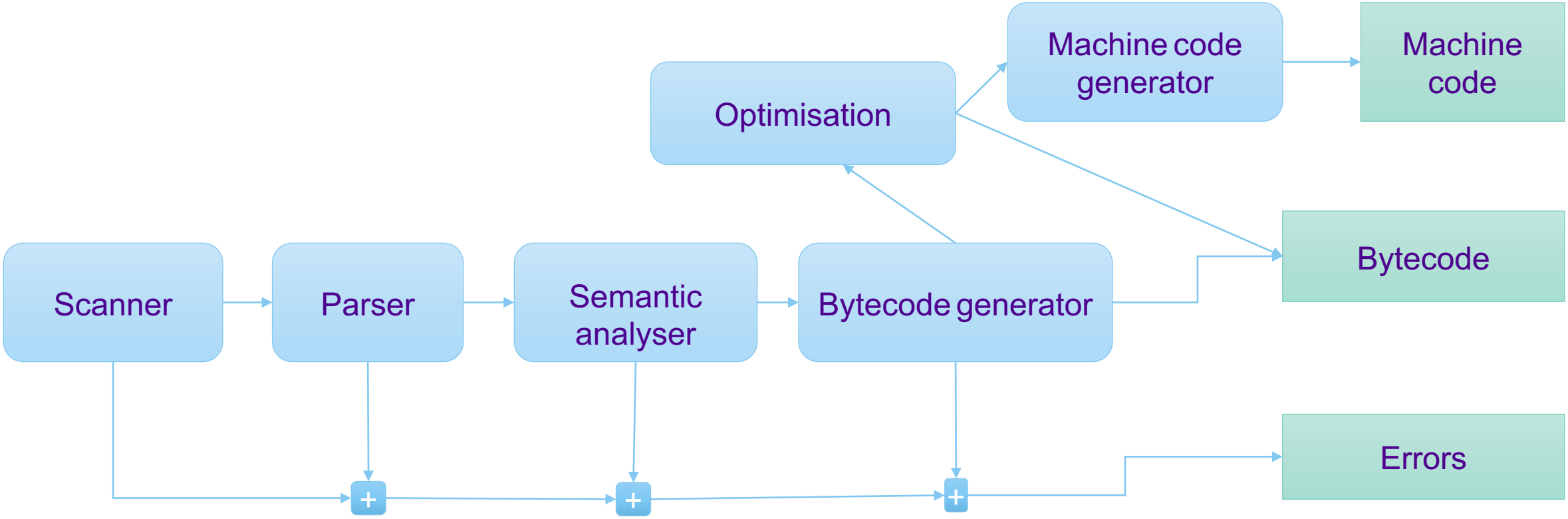
Pros and cons of pipes and filters architecture

- Pros
  - Complex information handling process can be divided pieces that are easier to handle.
  - Supports reuse: process units can be combined in several ways
  - Supports maintenance: processing unit can be easily changed.
  - Supports concurrency and distribution.
- Cons
  - Does not fit interactive systems (some exceptions like search engine)
  - Information interpreting may cause performance problems
  - Error handling may by difficult

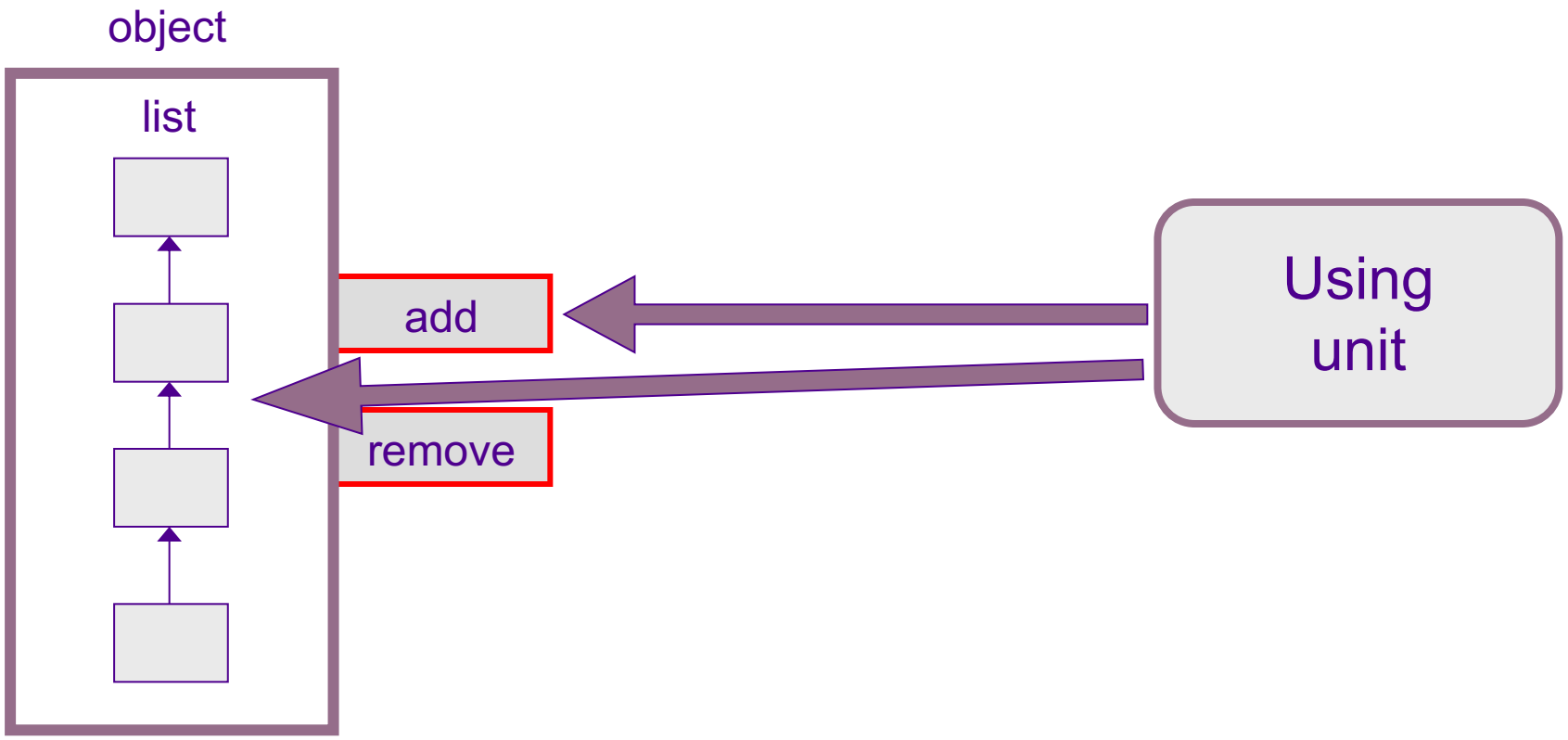
Error handling

- One possibility: a specific error flow, normal data and errors are separated (like Unix command line)
- On error side own filter that acts accordingly.

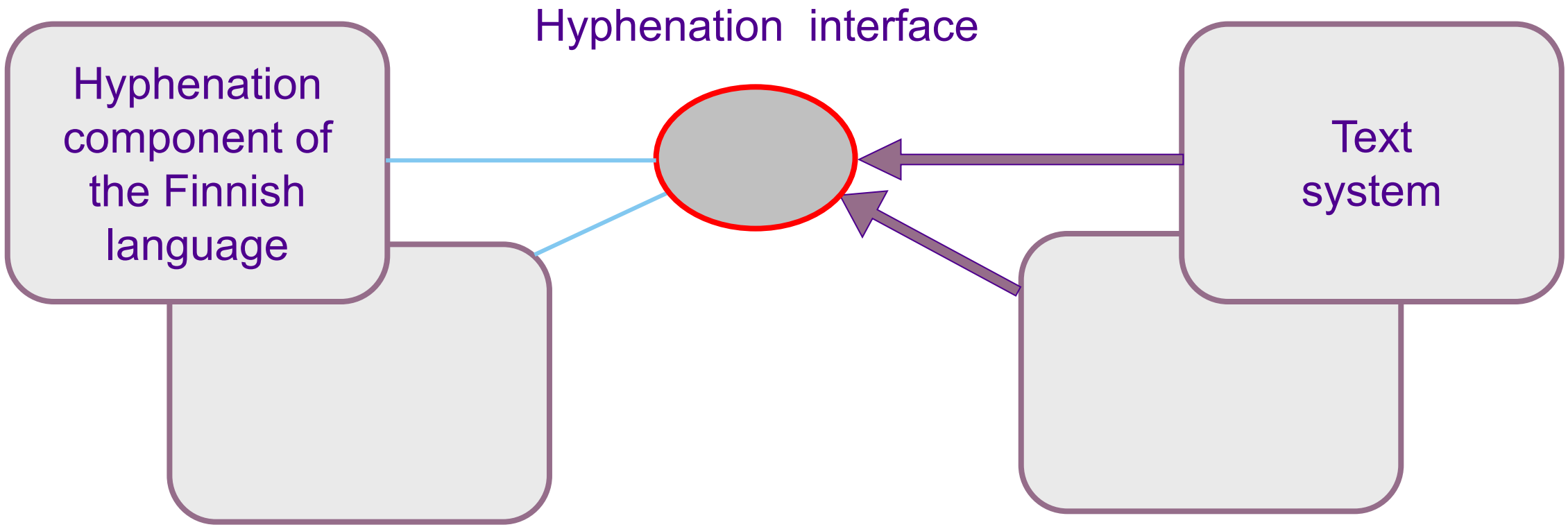
Compilation with error handling



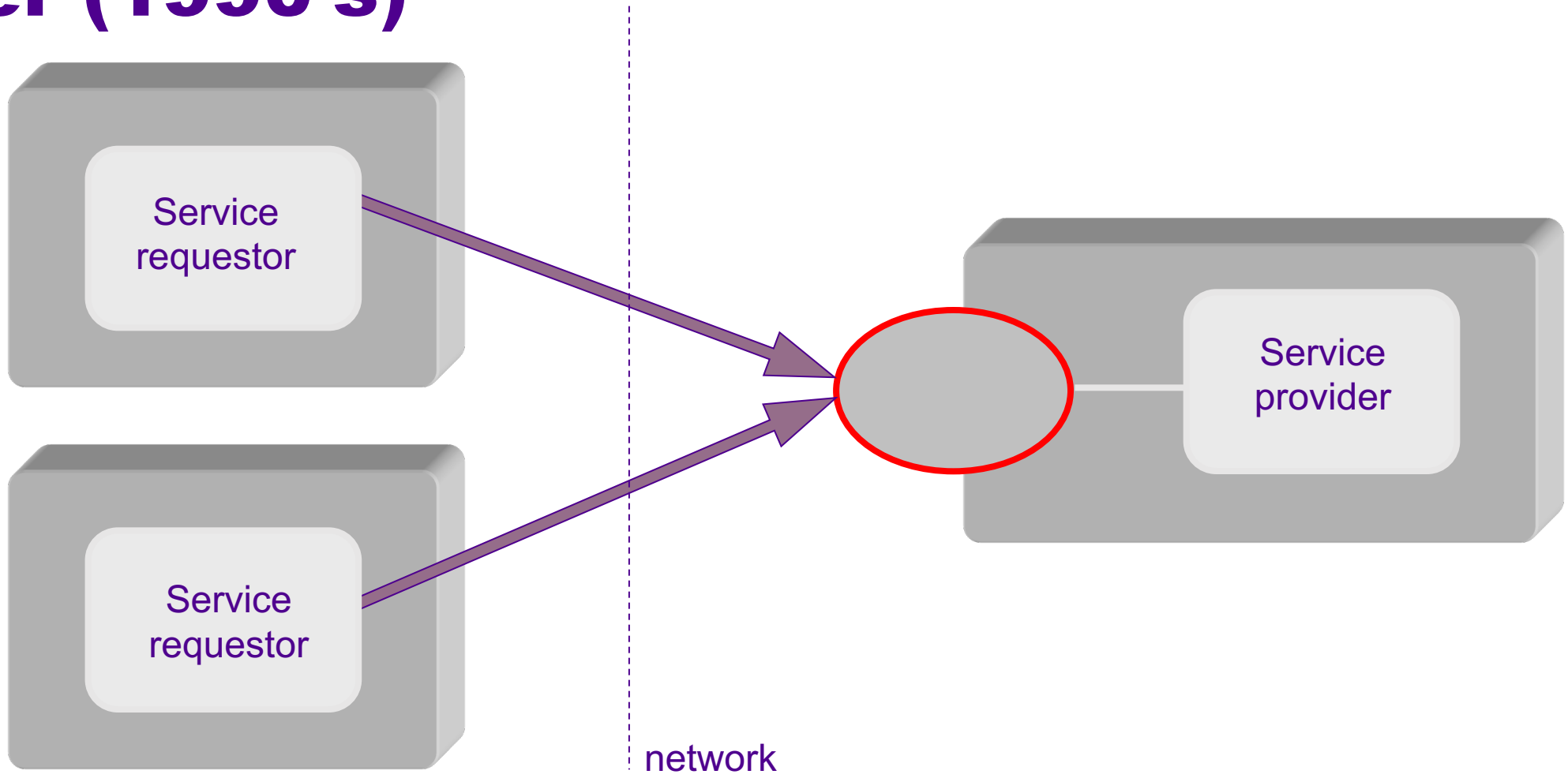
Service-based architecture styles: Objects (1970's)



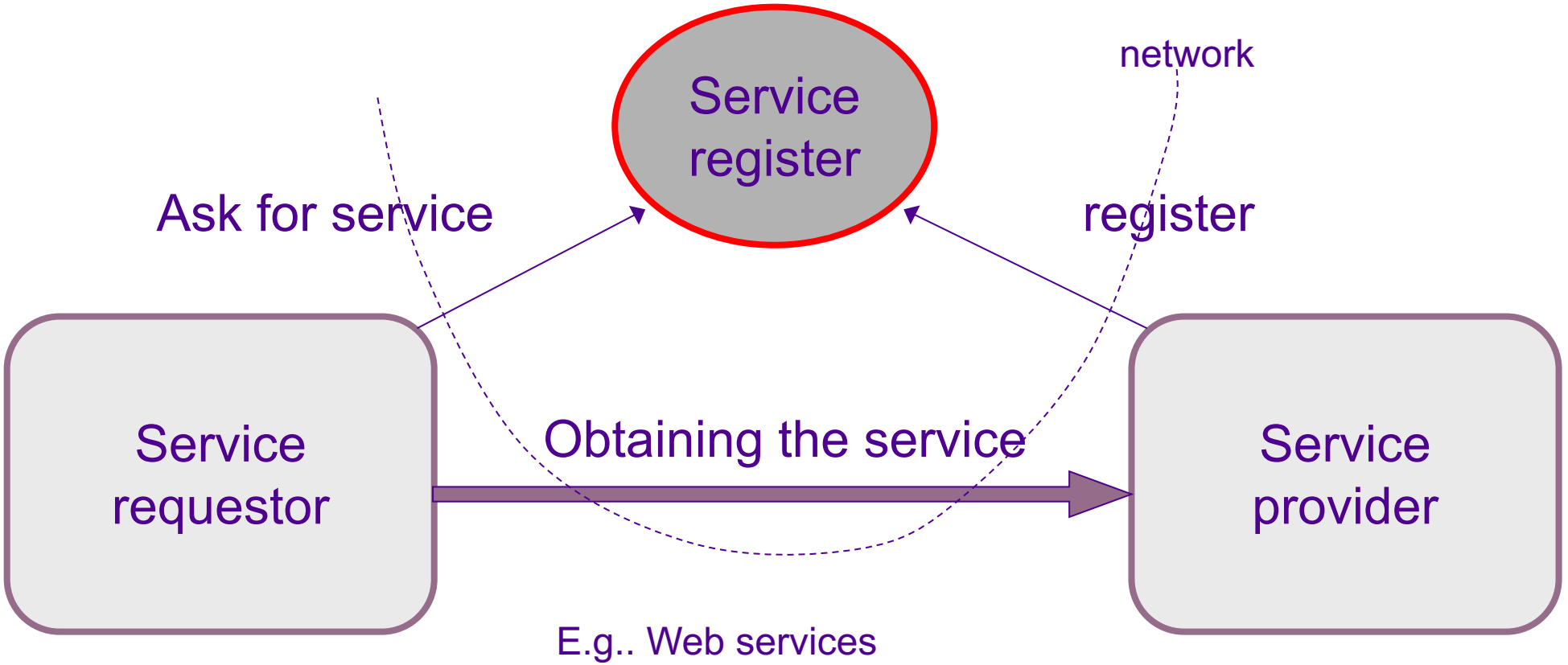
Service-based architecture styles: Components (1980's)



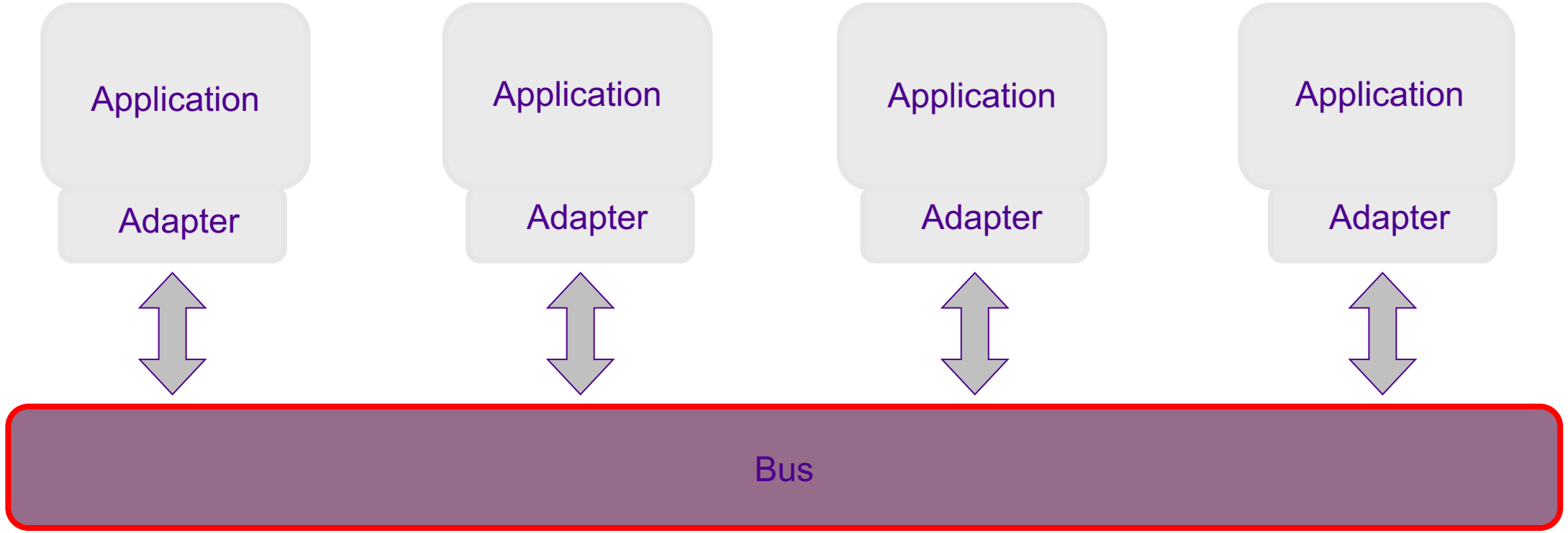
Service-based architecture styles: Client-server (1990's)



Service-based architecture styles: Service-Oriented Architecture (2000's)



Service-based architecture styles: Enterprise Service Bus, ESB



- Idea: message routing depends on the message -> there will not be dependencies between applications

Spirit of SOA

- Independent, independently maintained and managed components
- Composing software from these components
- No strict connections between components, using messages in communications
- Easy to distribute functions of components

Microservices

- Variant of service-oriented architecture.
- Fine-grained services connected together with lightweight protocols.
- Improves modularity
- Easier to understand, develop and test
- Philosophy: “Do one thing and do it well”.
  - Old Unix systems had this same idea, but the call structure was defined by the caller (a pipe)

Microservice definition

No final consensus regarding the properties of microservices. The properties often include:

- Services are processes communicating with each other using light-weight protocols (HTTP). However, other kinds of communication mechanisms are allowed (e.g. shared memory).
- Services might also run within the same process.
- Services should be independently deployable.
- Services are easy to replace.
- Services are organised around capabilities.
- Services can be implemented using different SW technologies (programming language, database), hardware and software environment, depending on what fits best.
- Services are small in size, messaging enabled, bounded by contexts, autonomously developed, independently deployable, decentralized, and built and released with automated processes.

Microservices-based architecture, pros

- Naturally enforces a modular structure.
- Lends itself to a continuous software development process.
- Provides characteristics that are beneficial to scalability.
- Each service is easy to test

Microservices-based architecture, cons

- Services form information barriers.
- Inter-service calls over a network have a higher cost in terms of latency and message processing time than in-process calls of a monolithic service process.
- Testing and deployment are more complicated.
  - Testing a single service is not the problem but whole construction
- Moving responsibilities between services is difficult.
- Can lead to too many services when the alternative may lead to a simpler design.

Conclusions, part 1

- Tier / layer architecture divides system on conceptual levels.
- Pipes and filters –architecture divides system by working phases.
- Service-oriented systems have been developed from object oriented programming towards service bus solutions and cloud systems.

Client-server architectures

- Client-server architecture: the system consists of servers controlling resources and providing services and clients needing services.

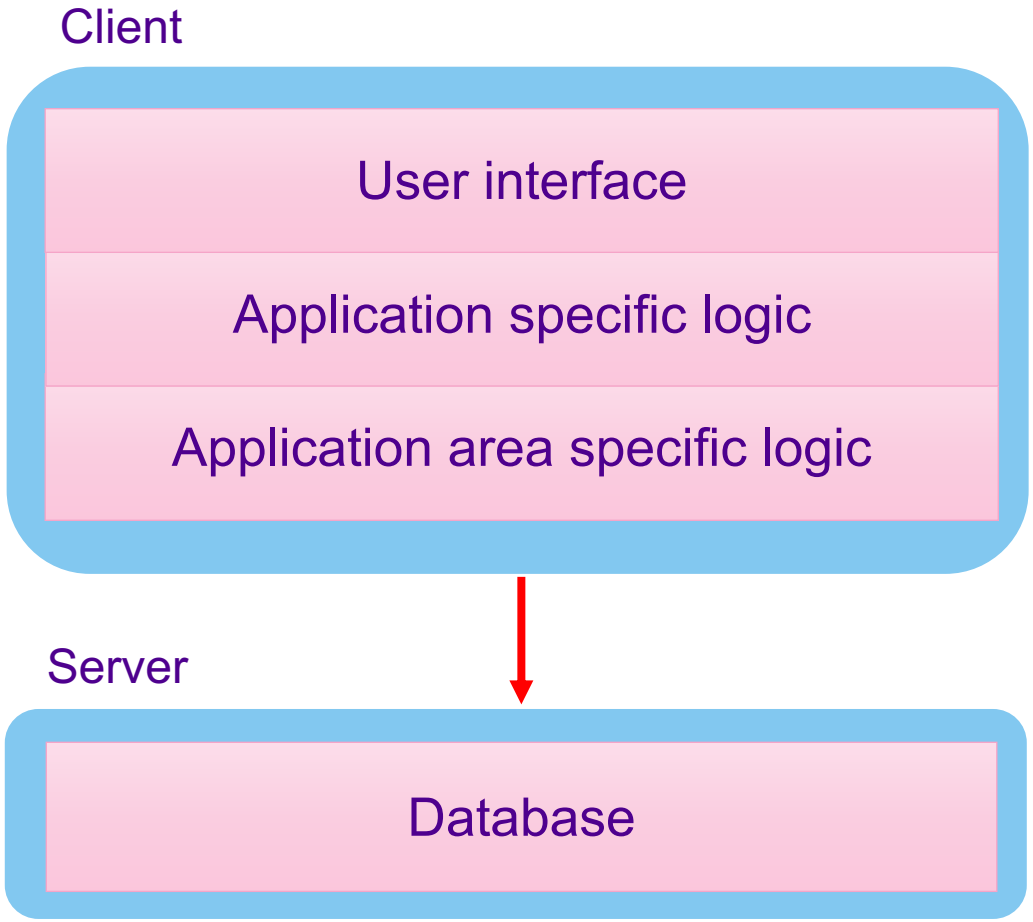
Client-server architectures

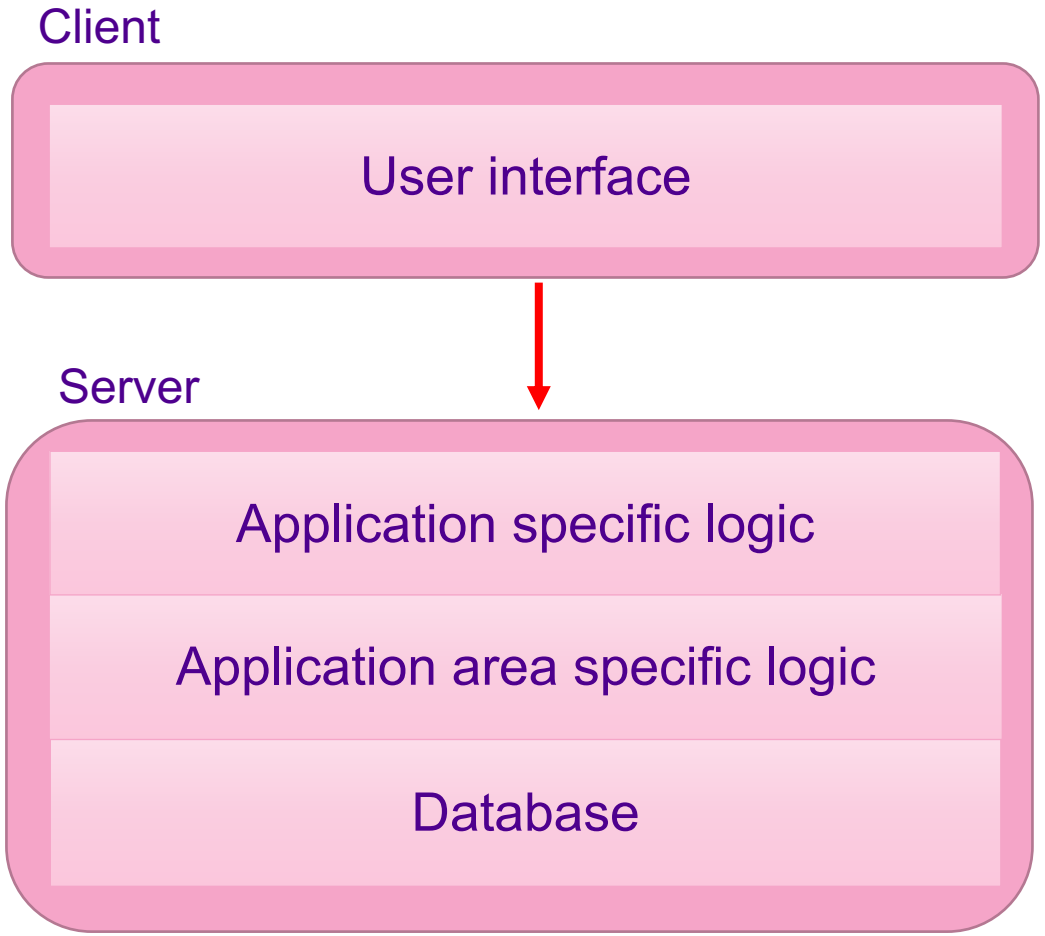
- Services are available in *sessions*; services belonging together are controllably given during a connection as transactions.
- Clients and servers execute in their own processes, typically distributed.
- Clients are typically applications that do not know each other.
- Servers do not know clients.
- Servers typically manage a resource or data storage.

Examples

- Data storage servers
- Systems based on application servers
- Email programs (server – terminal program)
- Network application etc.
  - Common solution, REST interfaces and users
- Most of the current applications and devices are (somehow) following client-server model.

Data storage and application servers





Pros and cons of client-server architectures

- Pros:
  - Eases controlling the common resource (security)
  - Eases maintenance and adaptability (changing the server)
  - Good technological support
- Cons:
  - Performance problems due to network traffic
  - Server-centric: sensible to failure on critical server
  - Exception handling

Peer to peer (P2P)

- Specialisation from client-server.
- Clients are also servers; equal = peers.
- Sharing resources, information, computing power, channel bandwidth, etc.
- Error-sensitiveness decreases, resource sharing makes it possible to solve bigger problems.
- Hybrids exist: centralised server and with P2P connections between clients.

Peer-to-peer networks

- Peer-to-peer (P2P) network
  - consists of autonomous peers
  - is a self-organizing system,
    - purpose: the usage of distributed resources
    - avoiding centralised services.
- P2P is also used to hide the "culprit"
  - In this course, we focus on technology (and legal usage)

Peer-to-peer Networks

- Three requirements for future Internet systems
  - scalability
  - security, and reliability
  - flexibility, and the Quality of Service.
- C/S has some potential problems:
  - Centralised server and its network traffic are bottlenecks.
  - It is easy to attack against the server.

P2P: New paradigm

- P2P is not just file sharing.
- P2P is a new paradigm for distributed systems.
  - coordination becomes co-operation
  - centralised becomes decentralised
  - participation provides incentives.
- P2P used to have a promising future

P2P application areas

- Storage capacity, file distribution systems and technologies (e.g. Torrent)
- Computing power, distributed execution, e.g. Bitcoin (Seti at home)
- Botnets
- Communication:
  - Skype (old version), centralised server, voice with P2P
  - Devices and machines communicating with each other.

Client-server vs. Peer-to-peer

Client-Server	Peer-to-peer
Centralised resources.	Resources distributed to peers.
Clients communicate via the server.	The peers communicate directly with each other (but not necessarily directly to the desired resource).
Clear roles between clients and servers.	Peers operate in both user and producer roles.
The communication network is a star, the client knows the server address.	Communication connections directed by an overlay network.

Overlay Networks

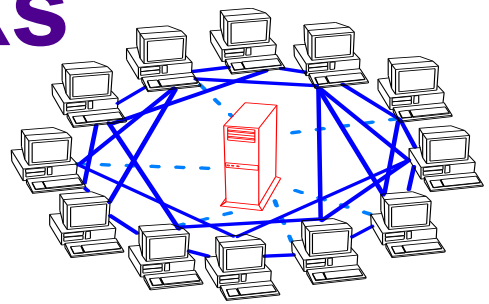
- The overlay network defines the communication links and addresses.
  - Overlay network = the logical network built on another network (physical or overlay)
- Deterministic overlay network.

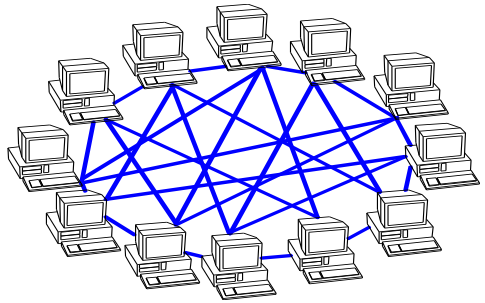
  ⇒Structured peer to peer network.
- Non-deterministic overlay network.

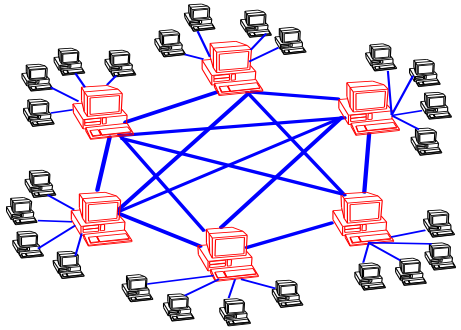
  ⇒Unstructured peer to peer network.

Unstructured P2P Networks

- Centralised
  - The central entity is responsible
  - for the addresses and the index.
- Pure P2P
  - No central entity.
- Hybrid
  - Dynamic central node.







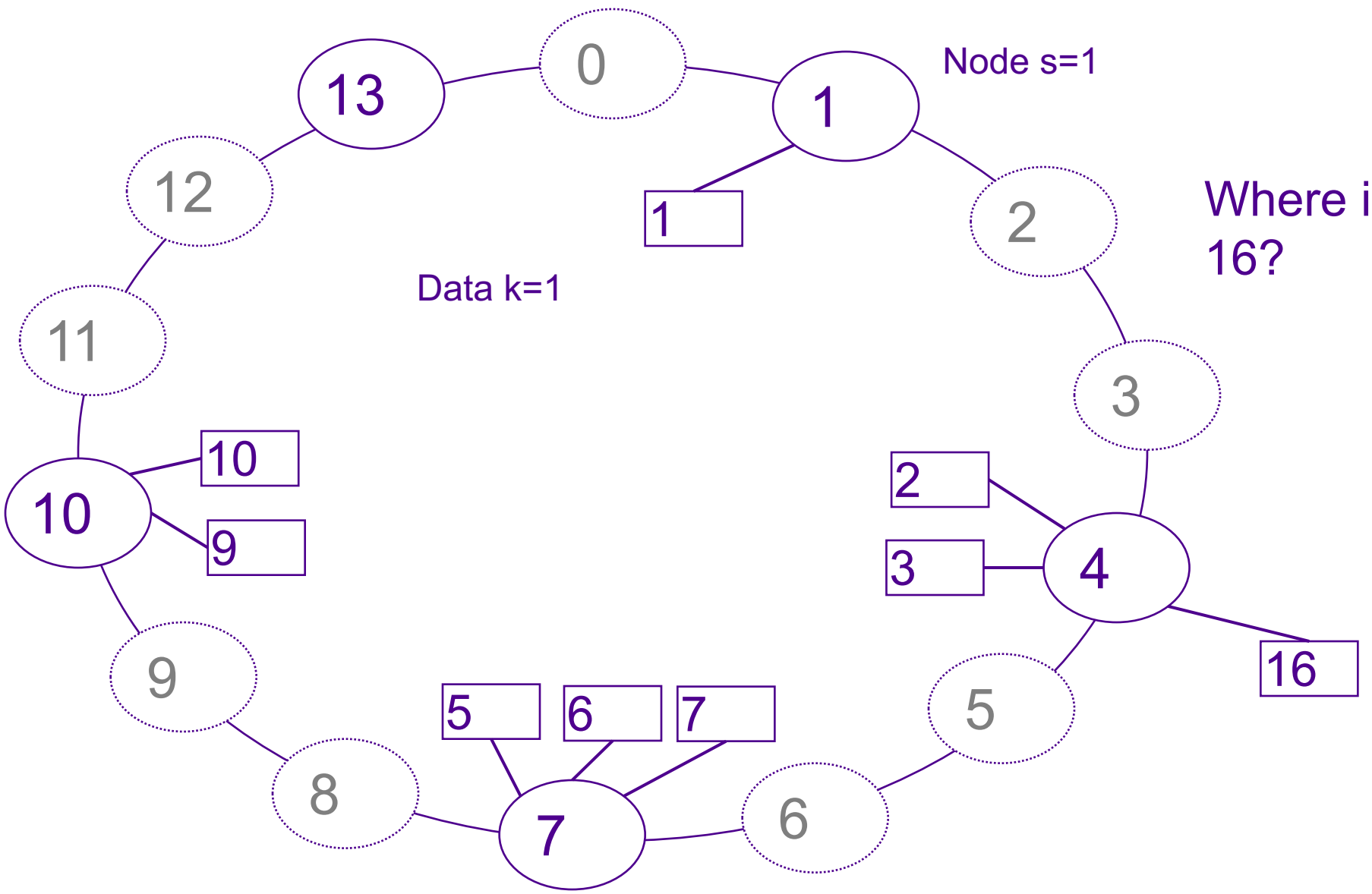
Structured P2P networks

- Goals
  - self-organising network
  - content-addressable data store
  - usually based on a decentralised hash table (DHT)
  - a distributed, scalable and fault-tolerant directory
  - search faster than in unstructured systems, typically O(log N).

Structured example: Chord

- The nodes form a ring.
- Every data item has an integer ID (key) *k.*
- Similarly, each node has id *s.*
- Each data element with key *k* is mapped to node *s*
  - smallest s, s ≥ k (mod n)
  - s = succ(k).
- When the application want to access data item *k* it calls function LOOKUP(k)
  - This returns the node that manages *k*, i.e. the network address of node *s.*

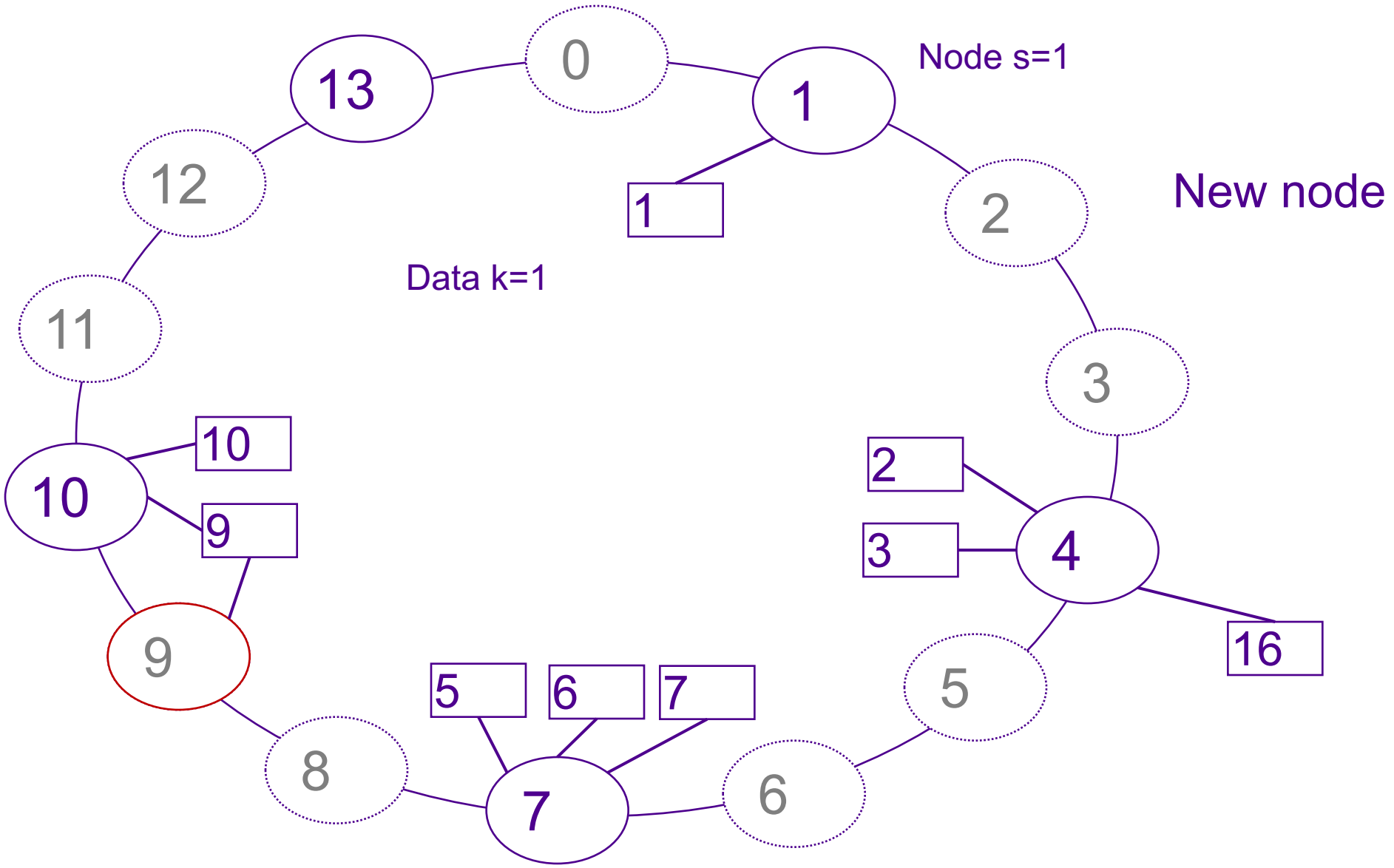
Chord


Smallest s, s ≥ k (mod n)
Node s=1
Where is 16?
Data k=1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
1
10
9
2
3
16
5
6
7

Chord

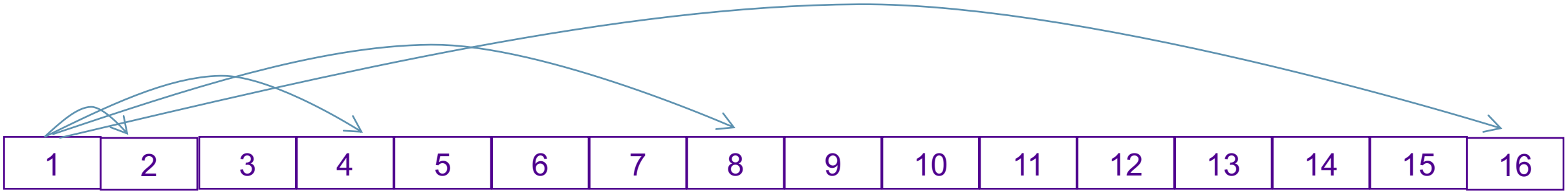
- Nodes know the other nodes.
  - Search O(log (N)); N is the number of nodes.
- Joining the ring:
  - Generate a random ID.
  - Function LOOKUP(ID) is called.
  - Register for node succ(ID) and its predecessor (node knows its predecessor and successor).
  - Transfer the relevant data to yourself.
- At departure, the predecessor and the successor are informed.

Chord


Smallest s, s ≥ k (mod n)
Node s=1
New node
Data k=1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
1
10
9
2
3
16
5
6
7

Optimisation

- Problem: Finding of the desired node may take a long time, because previous-successor chain has to be followed.
- This can be improved by adding shortcuts to a *finger table*.
- In the table of node *n*, item *i* contains address of the node *s* = succ(n + 2^{i-1})

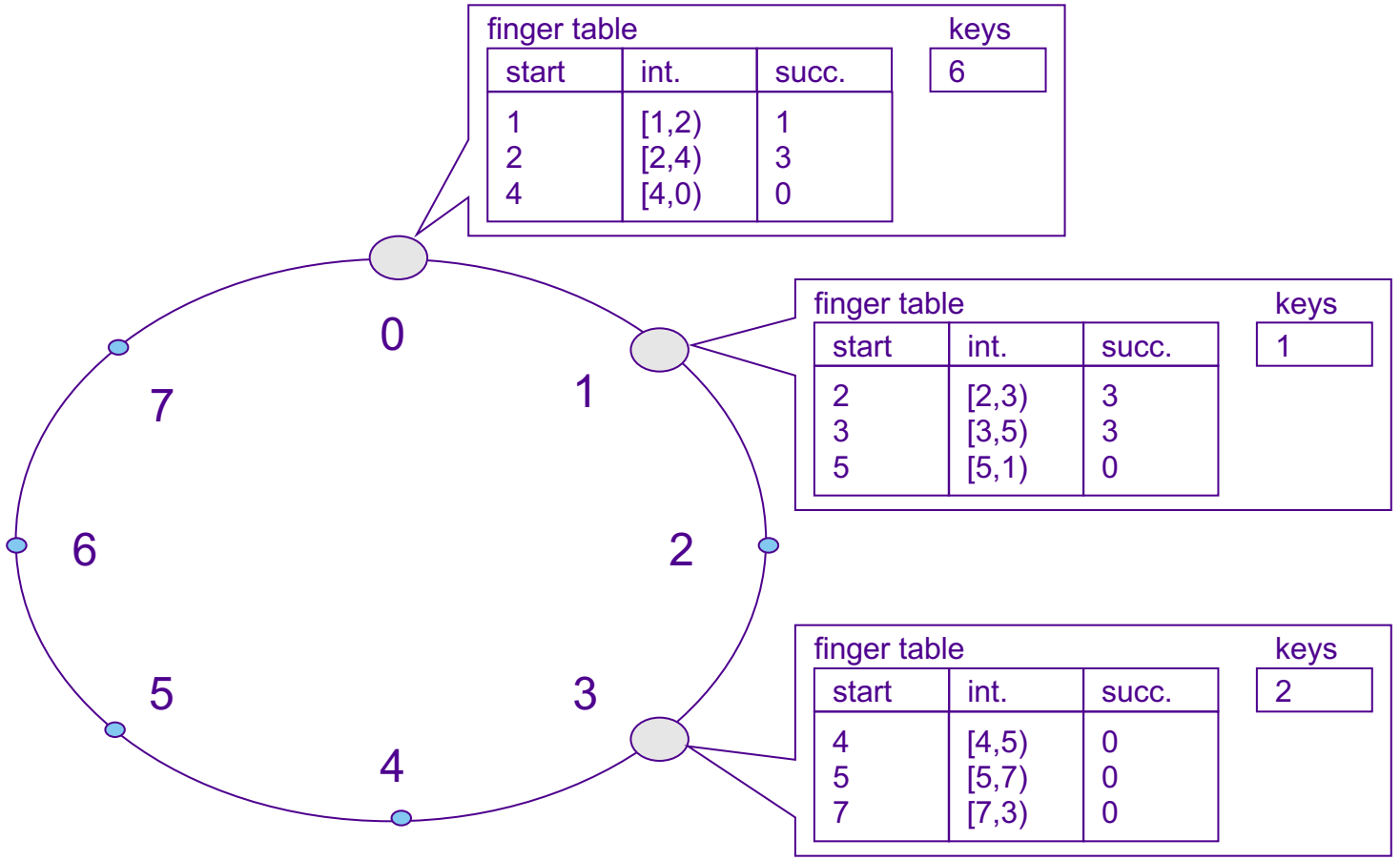


Challenges

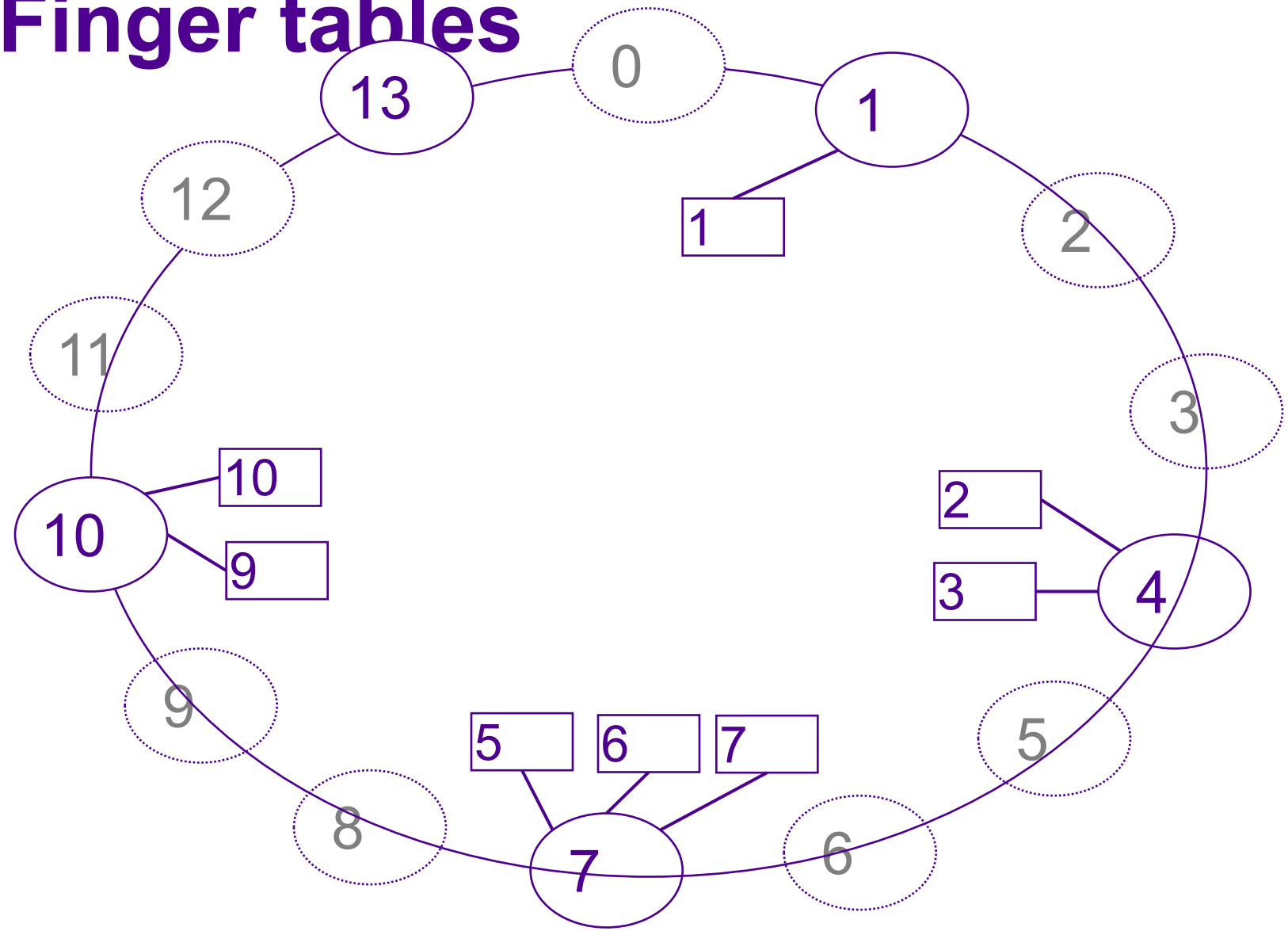
- In P2P systems, the data is distributed across different nodes => there is need for an efficient way of finding the data.
- Load balancing: load should be evenly spread between nodes.
- Decentralisation: no node may be more important or more critical than the others.
- Scalability: Adding capacity can simply be done by adding more nodes.
- Reliability: The information is always available.
- Flexibility, e.g., in terms of naming.

Finger Tables

www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/slides/lec03-chord.ppt



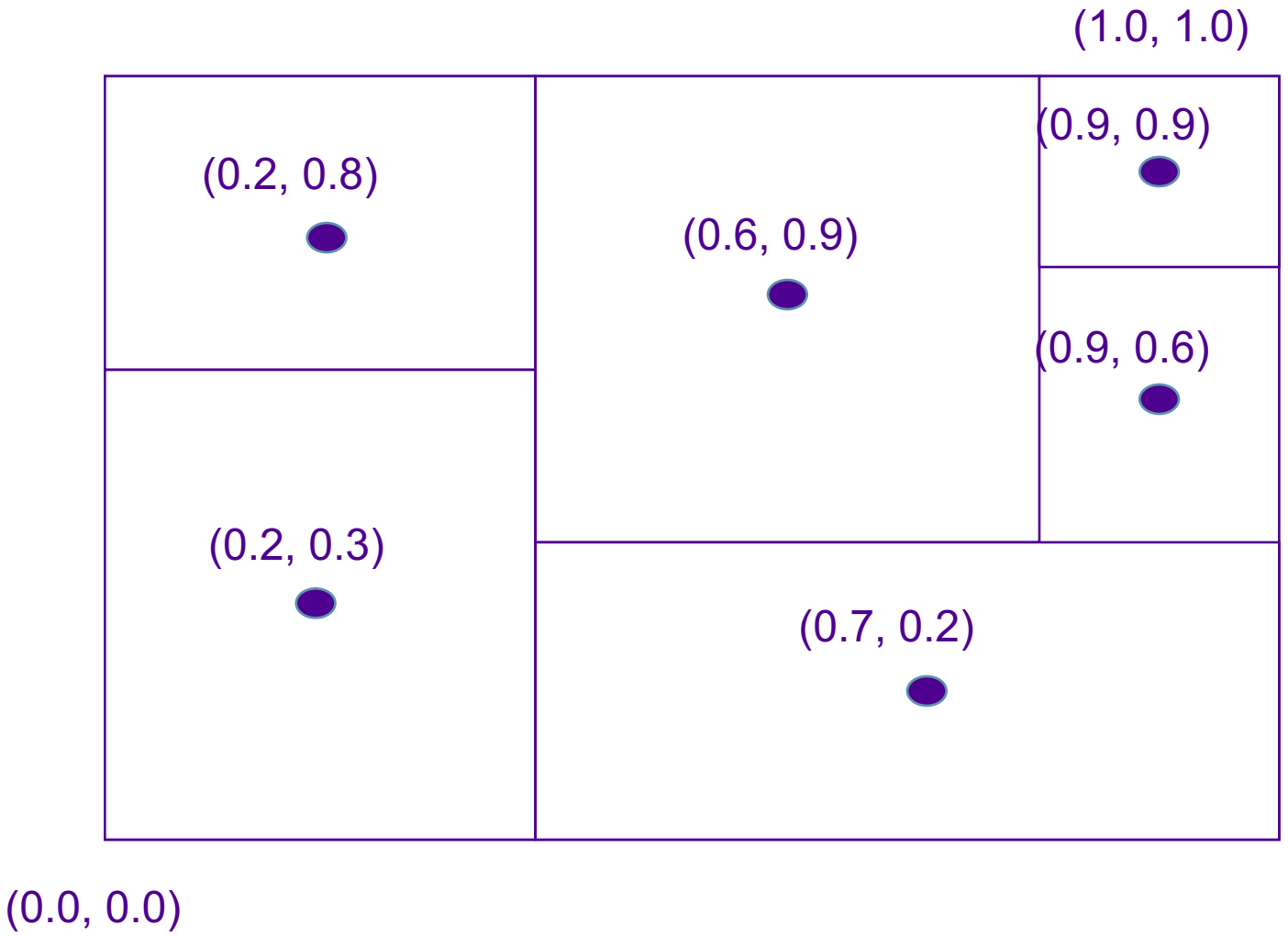
Finger tables



5	[5,6)	7
6	[6,8),	10
8	[8,12),	10
12	[12,20)	13

Distributed hash-table (DHT) example: CAN (Content Addressable Network)

- N-dimensional coordinate system is divided into zones.
- Hash key is a coordinate point in this space.
- Each zone is managed by a node with a coordinate.
- 2-dimensional example on the right.
- Adding dimensions shortens routes.



Centralised example: BitTorrent

- Centralised P2P file-sharing system.
- The name refers to the protocol, a peer or an official implementation of a peer.
- Peers of the system of peer download (and share) *pieces* of the file
  - piece size is typically 256 kB
  - piece is divided into 16 kB blocs.
- BitTorrent visualised: http://mg8.org/processing/bt.html

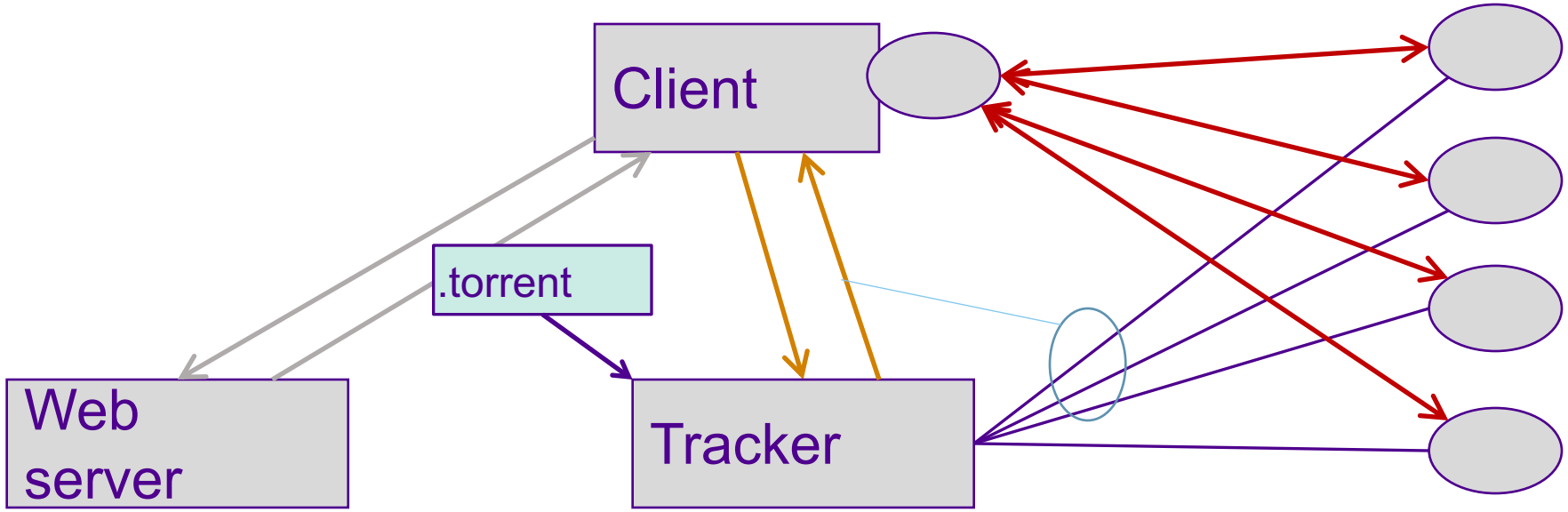
BT: Torrent

- Torrent defines a file-sharing session
  - Torrents are independent
  - Torrent is alive as long as it contains at least one copy of each piece of file.
- At first, the metadata file is requested from the Web server, i.e. Torrent. It includes
  - The name and size of the file to be downloaded.
  - SHA-1 fingerprints of the pieces.
  - Address of the tracker server.

BT: Tracker

- Tracker is a centralised resource that helps the peers to find each other
  - It keeps track of active nodes.
  - Those nodes have parts of the desired file.
- When starting up, peers get a subset of active nodes.
  - Standard algorithm returns a random set with a size of 50.

BitTorrent as an image



BT: Peer

- The client becomes active when it receives information from other peers.
- The peers copy pieces of file from each other.
  - To be precise, pieces are further subdivided into blocks that are requested in such a way that several requests are continuously in the queue.
  - The integrity of parts is checked using fingerprint stored in .torrent file.
- Visualisation of Bittorrent: https://mg8.org/processing/bt.shtml

BT: Peer

- Peer sends its state to the tracker every 30 s.
- Peers send to the known peers data
  - of the pieces they have and
  - which pieces they want get.
- How to choose which block to download
  - start-up: random first
  - normal mode: the rarest first
  - end game mode: all blocks are asked from all peers.

BT: Choking algorithm

- Choking is a temporary refusal to upload.
  - The idea is to deal with free rides, i.e., those who only download but never upload.
- At most four peers of the interested set are not choked (unchoke).
  - Interested peer is a peer that wants something from the current peer.

BT: Choking algorithm

- Unchoke is done in the following way:
  - Regular unchoke: every 10 seconds set of interested peers is ordered according their upload rate. Three fastest ones are unchoked.
  - Optimistic unchoke: every 30 seconds one of the interested peers is unchoked randomly.
- Without optimistic unchoke new peers could never get started.

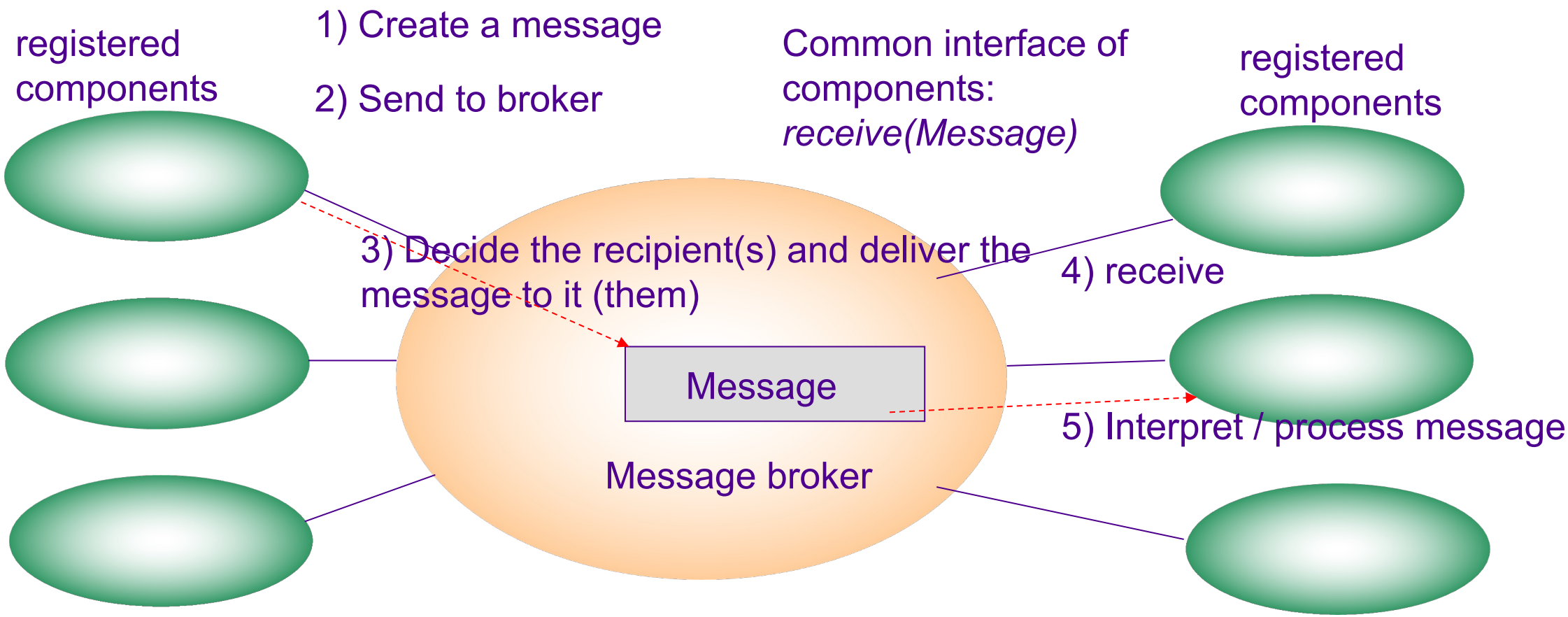
Pros and cons of P2P

- Pros:
  - Fault tolerance
  - sharing bandwidth and resources cause savings
  - easy to share resources, scalability
- Cons:
  - Security, safety
  - level of service
  - complex implementations (hybrid model)
  - New technology has decreased its usefulness.

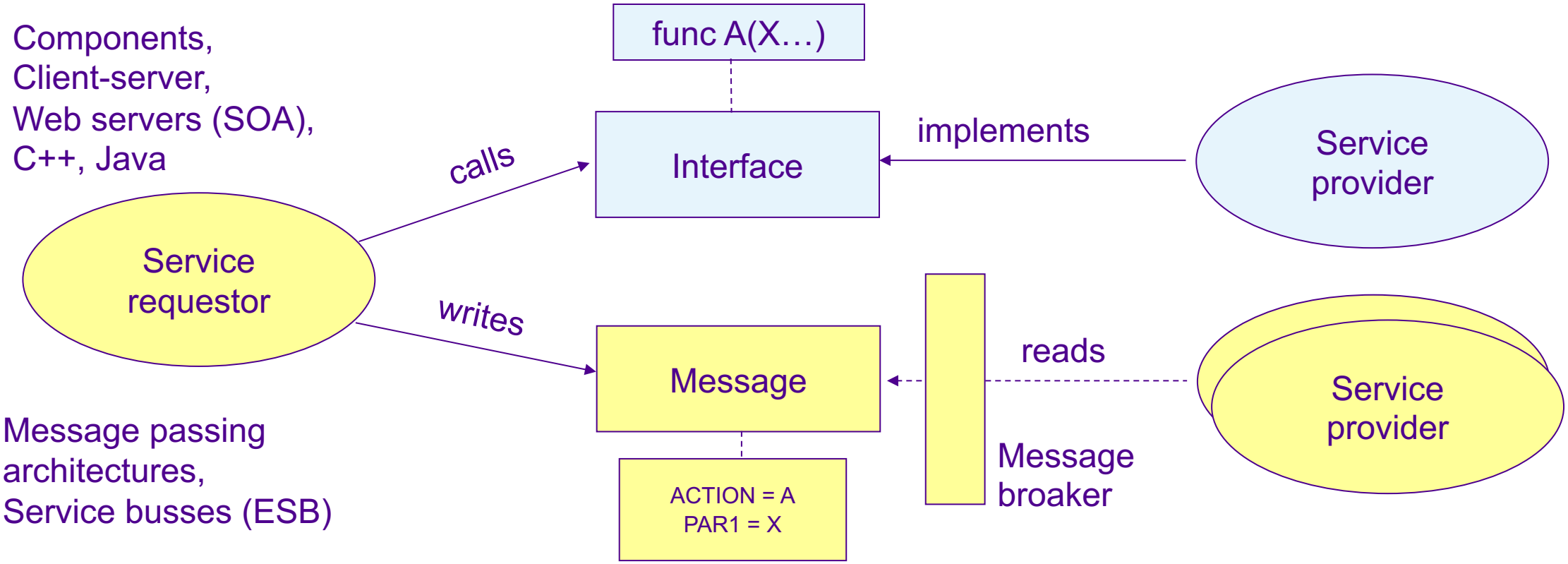
Message passing architectural styles

- Starting point
  - System consists of components communication with each other, possible distributed
  - Services of components are not known precisely in advance.
  - Components and number of them are not known precisely in advance
  - The quality of the information in systems is not known in advance

Message-passing architecture: basic idea



Interfaces vs. Messages

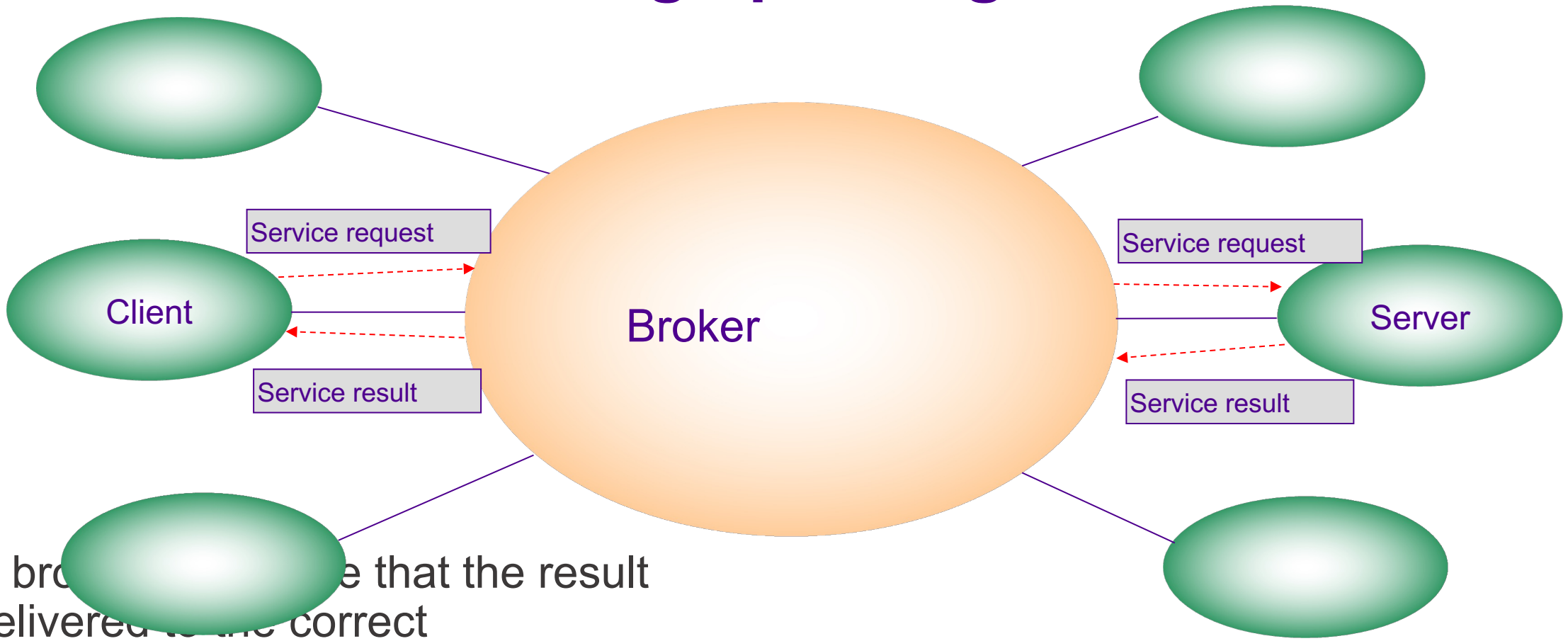


- Interface defines what is done and on which information, message can tell anything (what is done, who will do, which information)

Examples

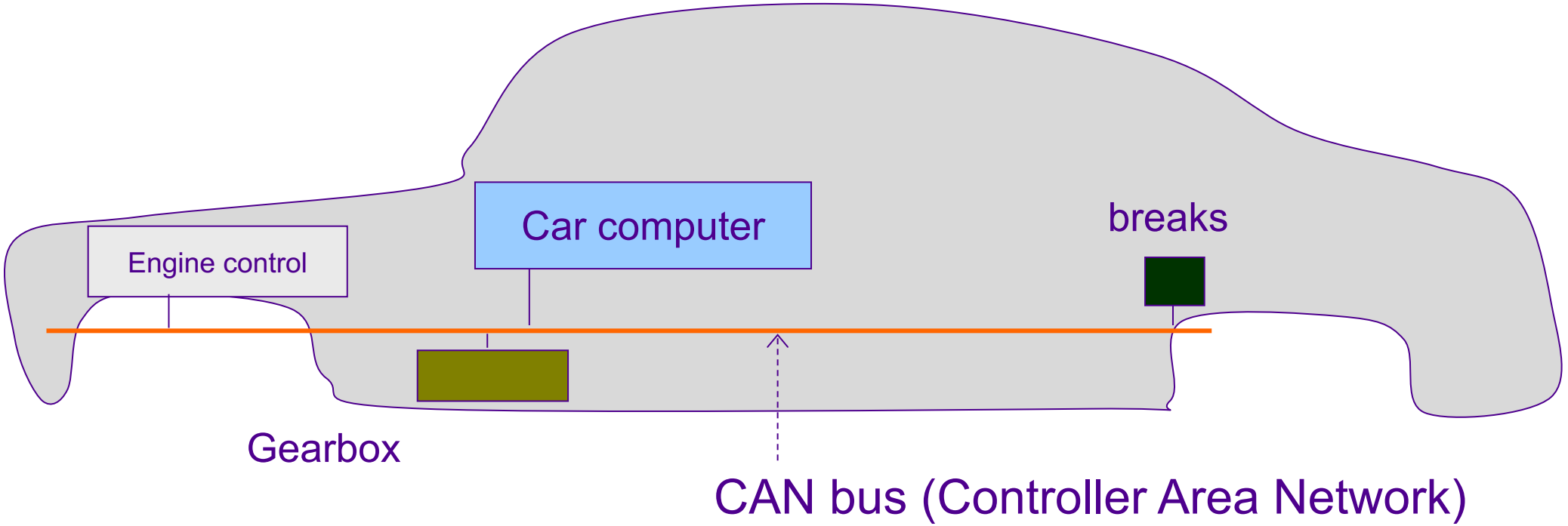
- Engine control systems
- IOT systems
- Multimodal systems, command-centric architectures
- Business management system of a company
- Generally: distributed systems, loosely-coupled systems

Service-based message-passing architecture

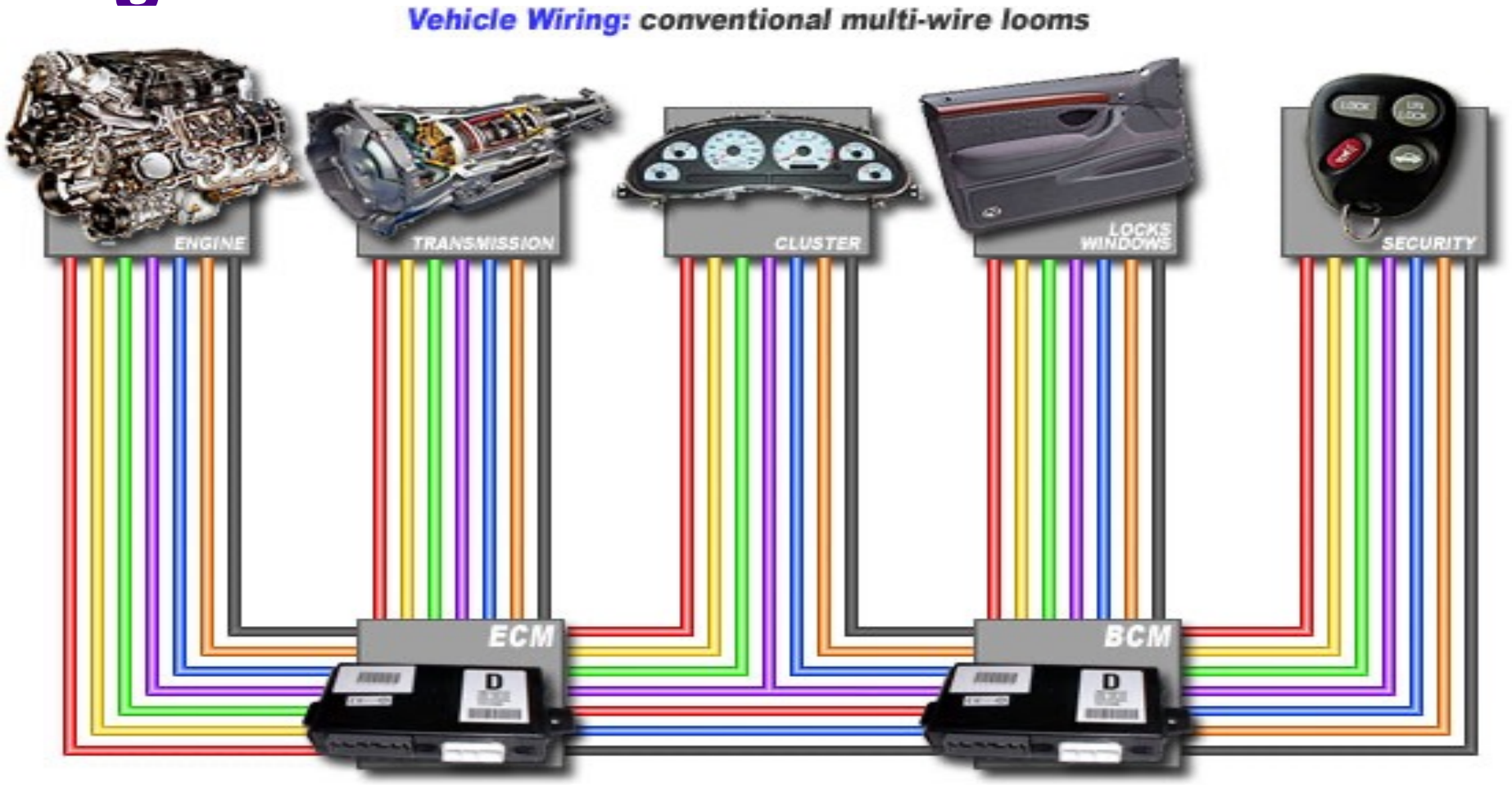


- The bro e that the result is delivered to the correct component

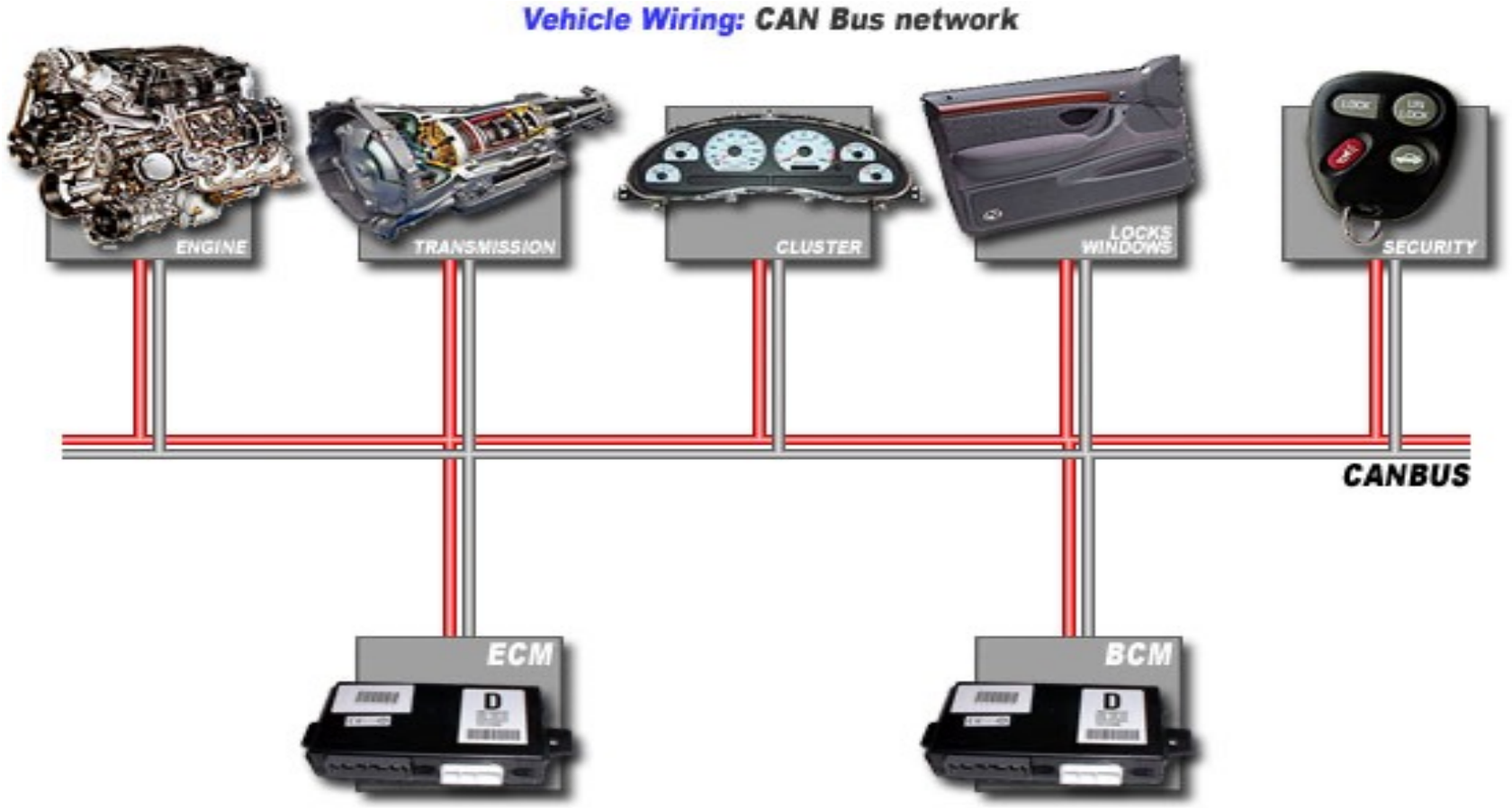
Example: Message bus in a car



Car wirings before bus



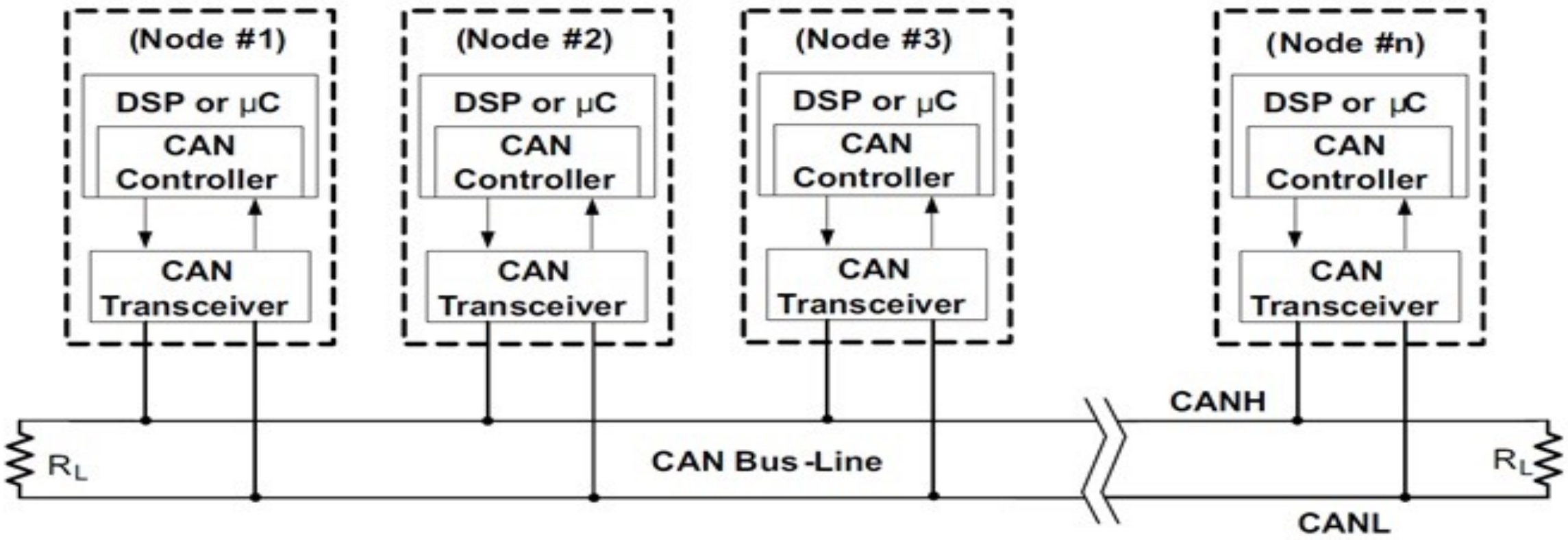
With a bus


Vehicle Wiring: CAN Bus network
ENGINE
TRANSMISSION
CLUSTER
LOCKS
WINDOWS
SECURITY
CANBUS
ECM
BCM

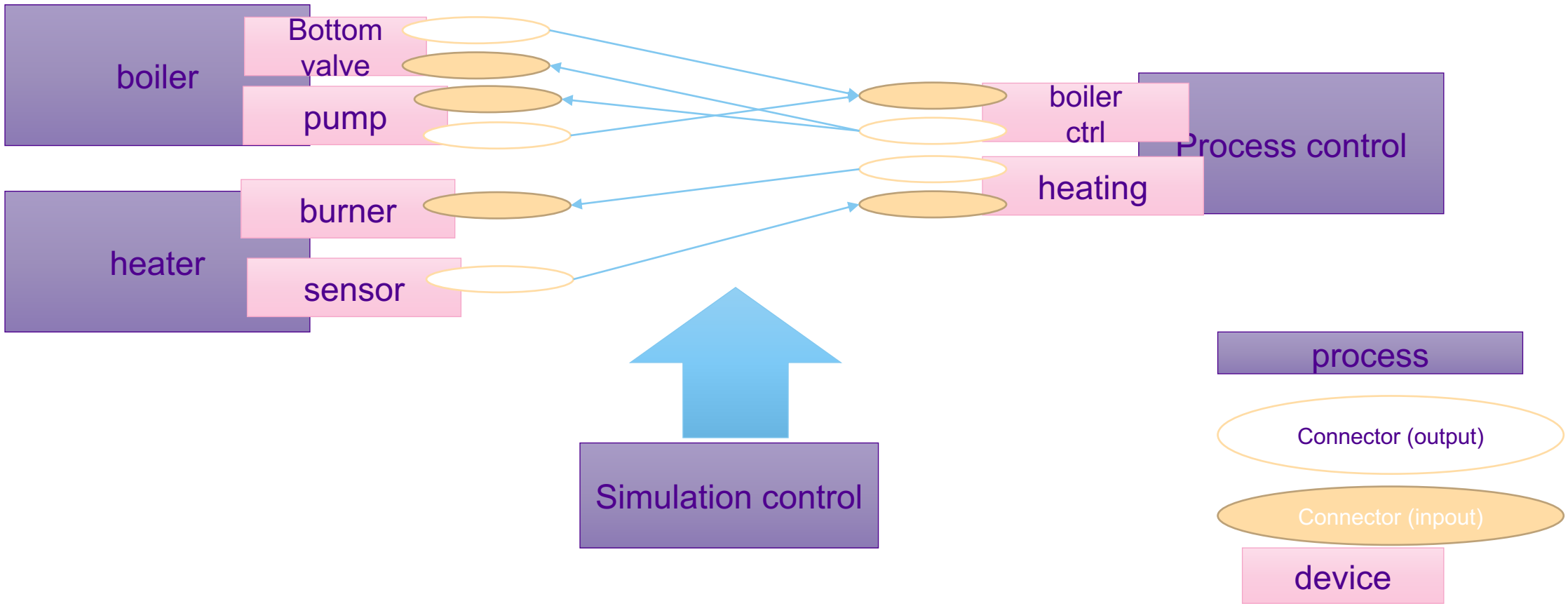
Message passing and messages

- The contents or format of a message essential
- Instruction, data, …
- To who
- Components have ID, priorities based on id’s
- Possibility to have multiple busses: entertainment, vehicle control, comfort

CAN bus


(Node #1)
DSP or µC
CAN Controller
CAN Transceiver
(Node #2)
DSP or µC
CAN Controller
CAN Transceiver
(Node #3)
DSP or µC
CAN Controller
CAN Transceiver
(Node #n)
DSP or µC
CAN Controller
CAN Transceiver
CANH
CANL
CAN Bus-Line
RL
RL

History: simulation environment of embedded software



History

- Internal implementation, broker in each process, devices register to a process (message sources & receivers) – these tell of their existence to the control.
- Simulation control creates simulation environment, controls is (e.g. time management), creates connections (according to simulation description)
- Message passing in all communication, messages.

Defining message types & contents

- Message passing architectures → messages
- It is important to define the structure, contents, possible error handling etc. of the messages.
- Different kinds of messages:
  - Event messages
  - Request – answer message,
  - Command messages (remote procedure)
  - Data messages (information delivery)

Pros of message passing architectures

- Easy to change, add and remove components or applications.
- Fault tolerant (e.g. if there is no receiver for a message), the message can be sent repeatedly
- Flexible system configuration
- Allows heterogenetic systems, application integration
- Allows both synchronic and asynchronic communication

Cons of message passing architectures

- Performance: writing and reading messages
- More difficult to implement, test and understand than traditional
- Some ”ordinary” things need special support (e.g. returning the result, synchronisation)
- Implicit connections created easily between units; virtually independent components hard to maintain and modify (especially if dependencies have not been documented)

Model-View-Controller (MVC) architectural styles 1

- Architectural solution for interactive systems separating the user interface from the application logic.

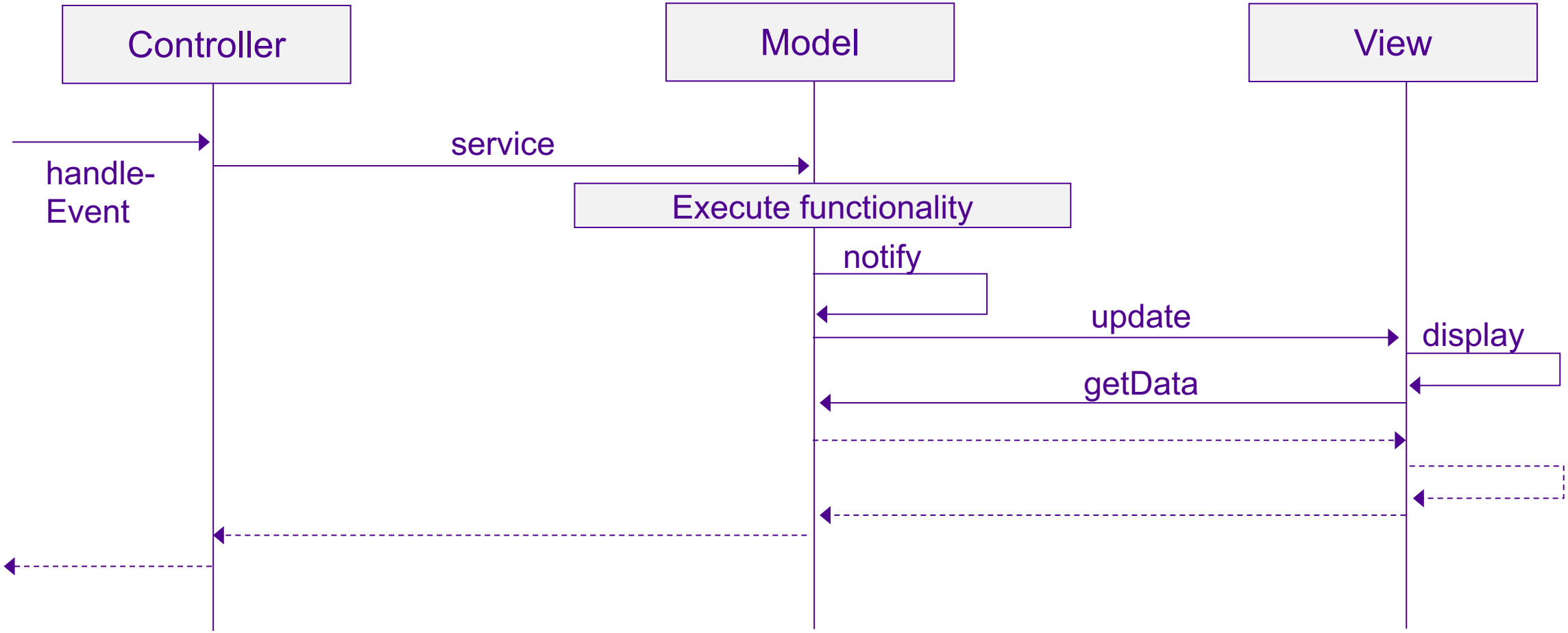
Model-View-Controller (MVC) architectural styles 2

- Starting points:
  - There should be possibility to offer different kinds of views from the state of the application.
  - The user interface should immediately reflect the changes of the application state.
  - The user interface should be easy to change
  - It should be able to transfer the application to another graphical platform.

Responsibilities

- Model
  - Offers logical functions and information of the application
  - Registers viewing components interested in the state of the application
  - Informs state changes to registered components
- View
  - Takes care of displaying the state on the display
- Control
  - Reads user commands
  - Changes the command to application functions

MVC interaction



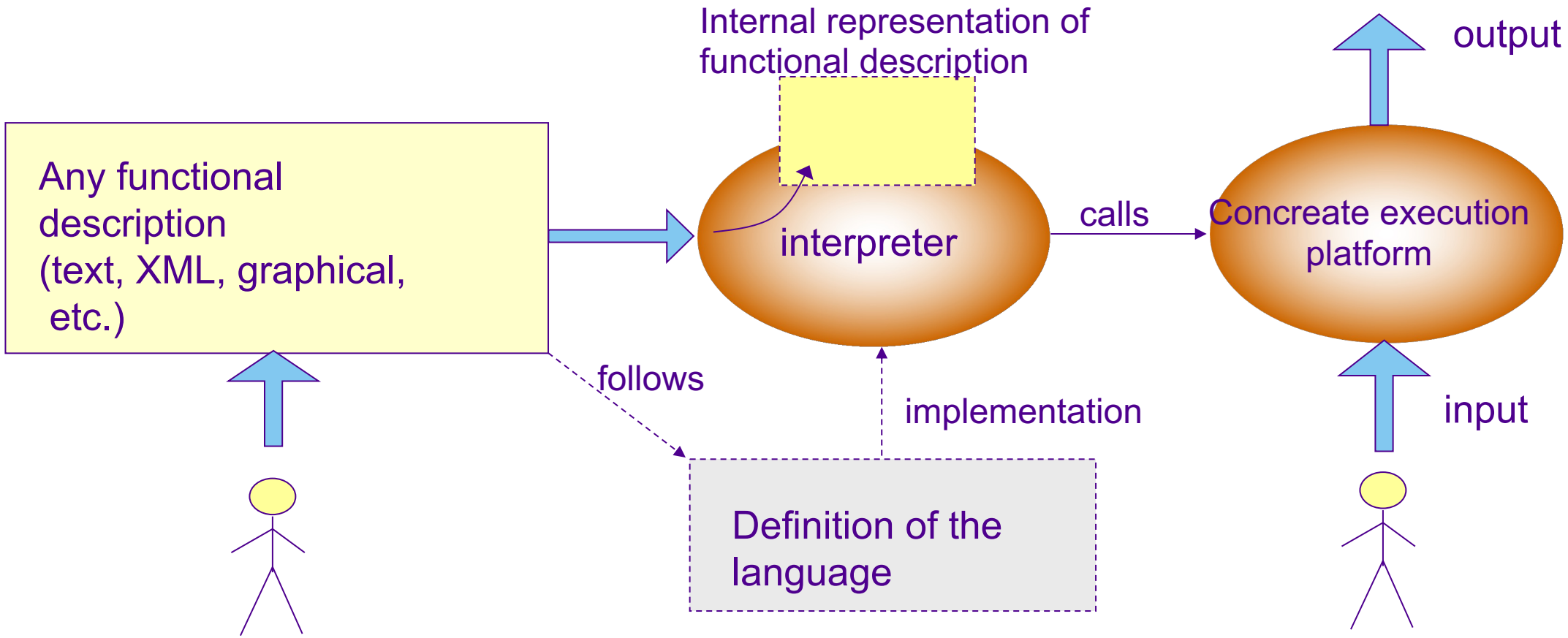
Pros and cons of MVC

- Pros
  - Easy to implement several views to the same data
  - All views are automatically synchronised
  - New views can be added to a running system
  - The layout of the interface can be changed with relative ease
- Cons
  - Possibly unnecessary requests to update the view
  - Metadata inquiries may increase execution time
  - For simple applications a lot of extra work

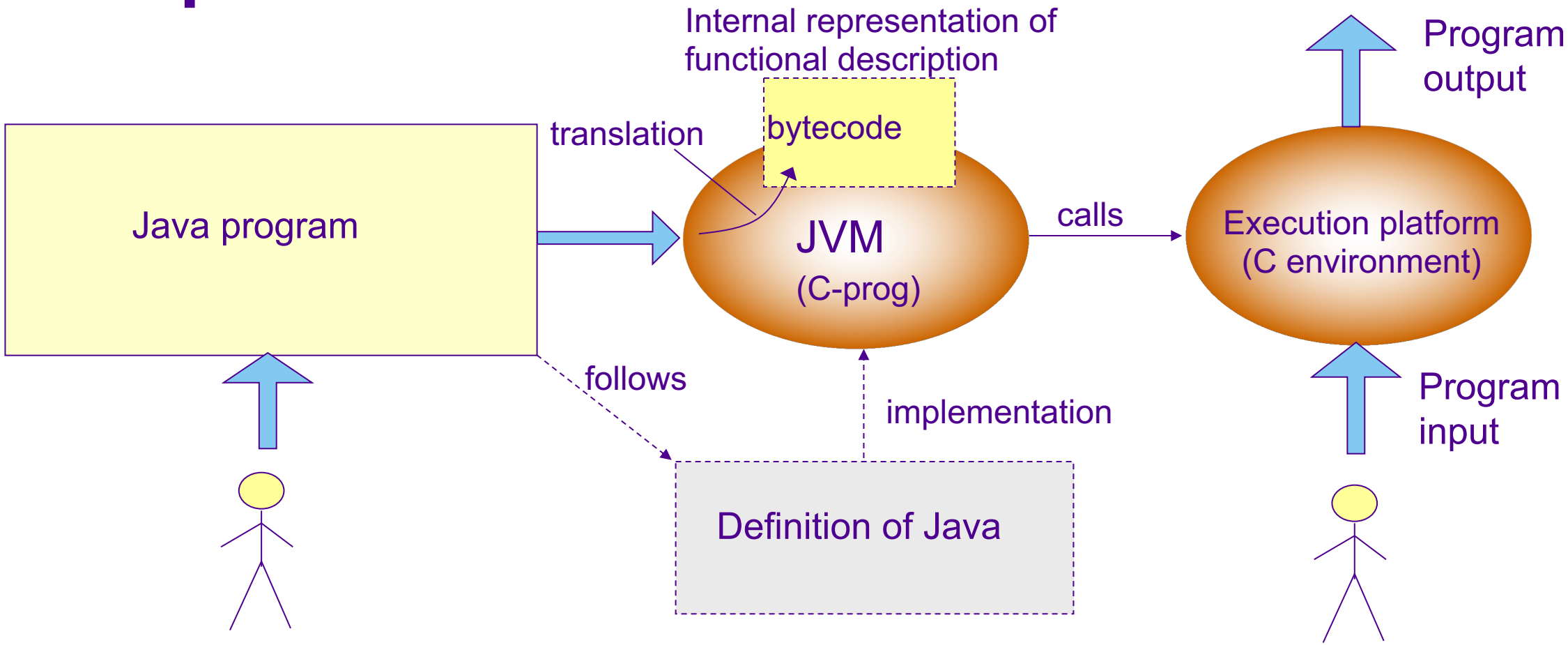
Interpreter architectural styles

- The need to give functional descriptions as input to the system. Examples:
  - Need do combine primitive functions in different ways that are not known in advance.
  - Need to separate a logical, abstract execution platform from concrete one (e.g. to make it easier to change the latter one)

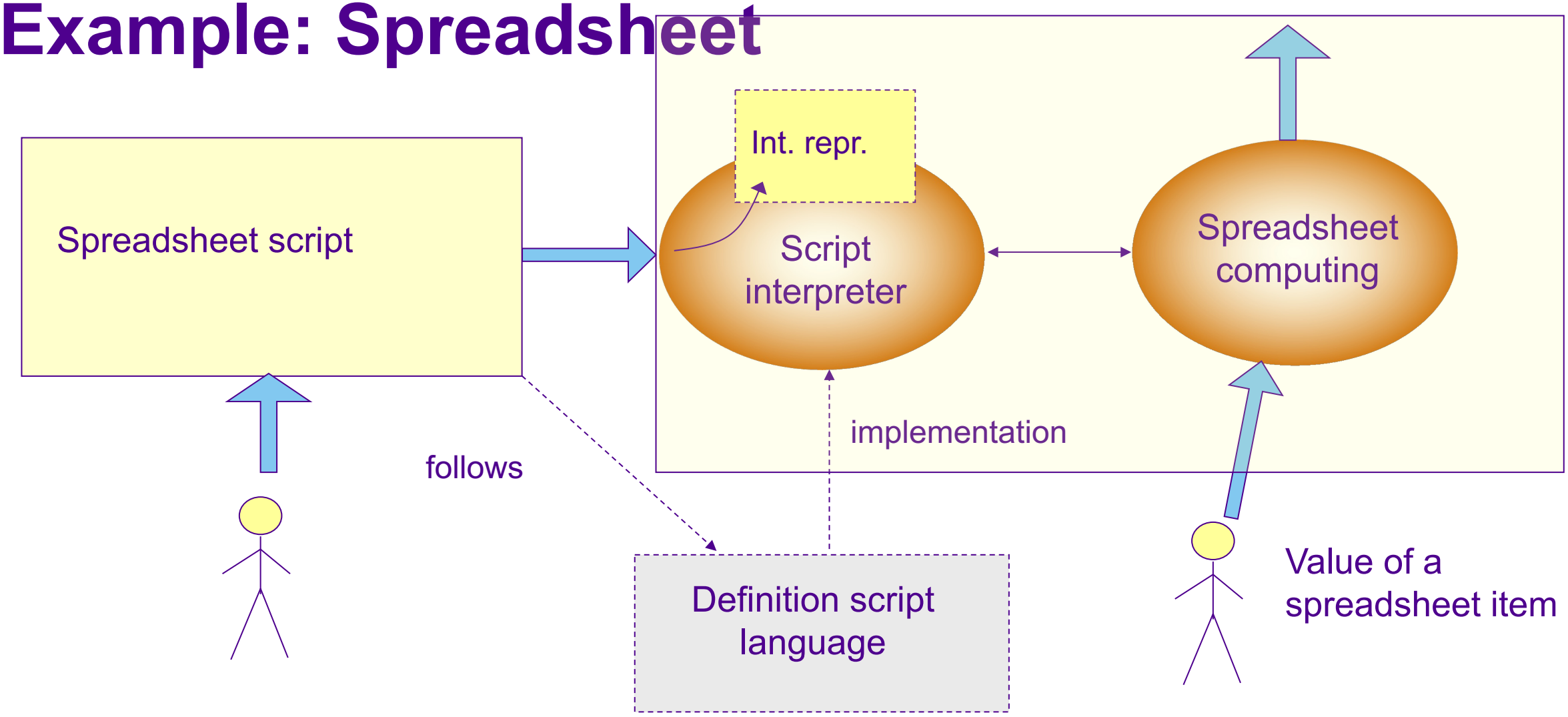
Basic ideas of interpreter architectural style



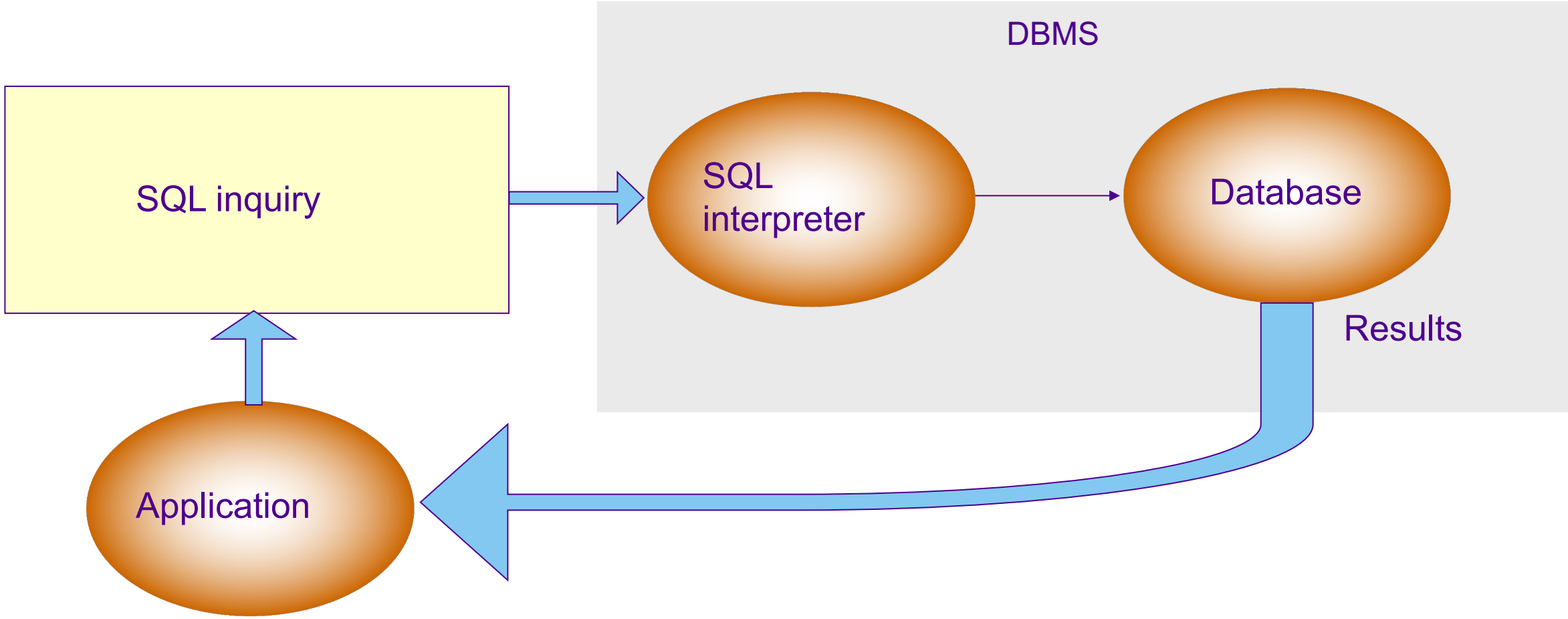
Example: Java



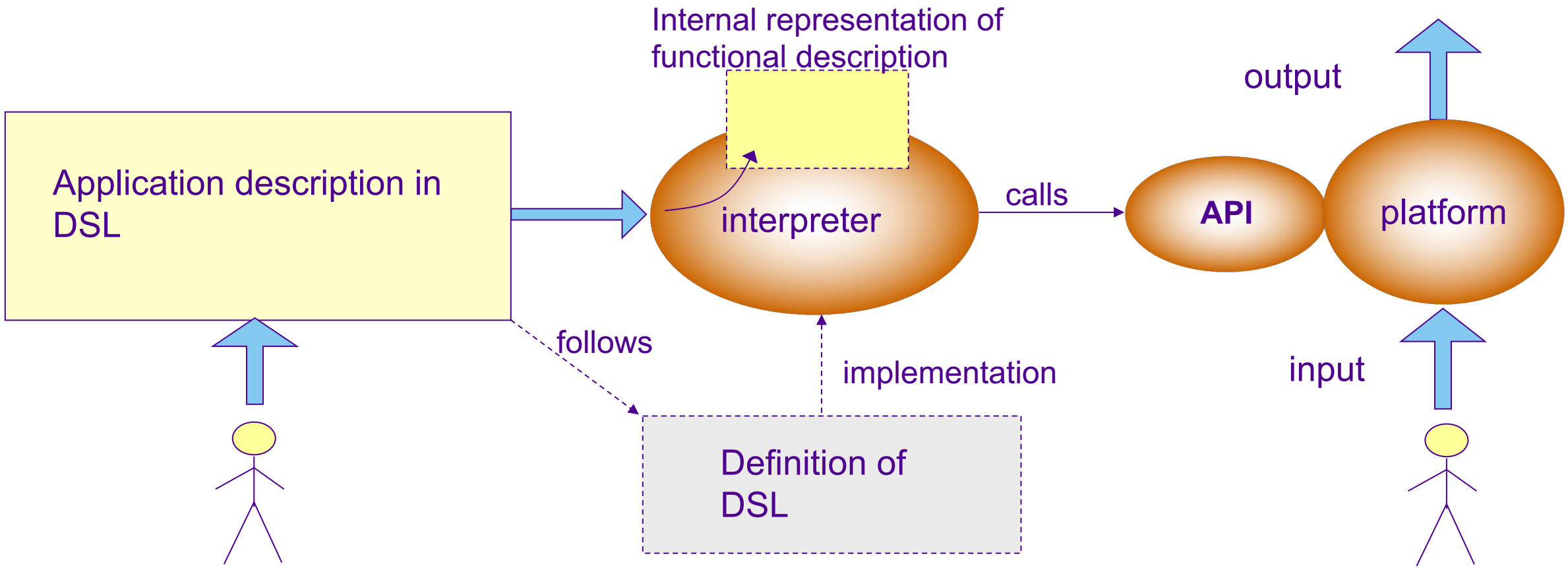
Example: Spreadsheet



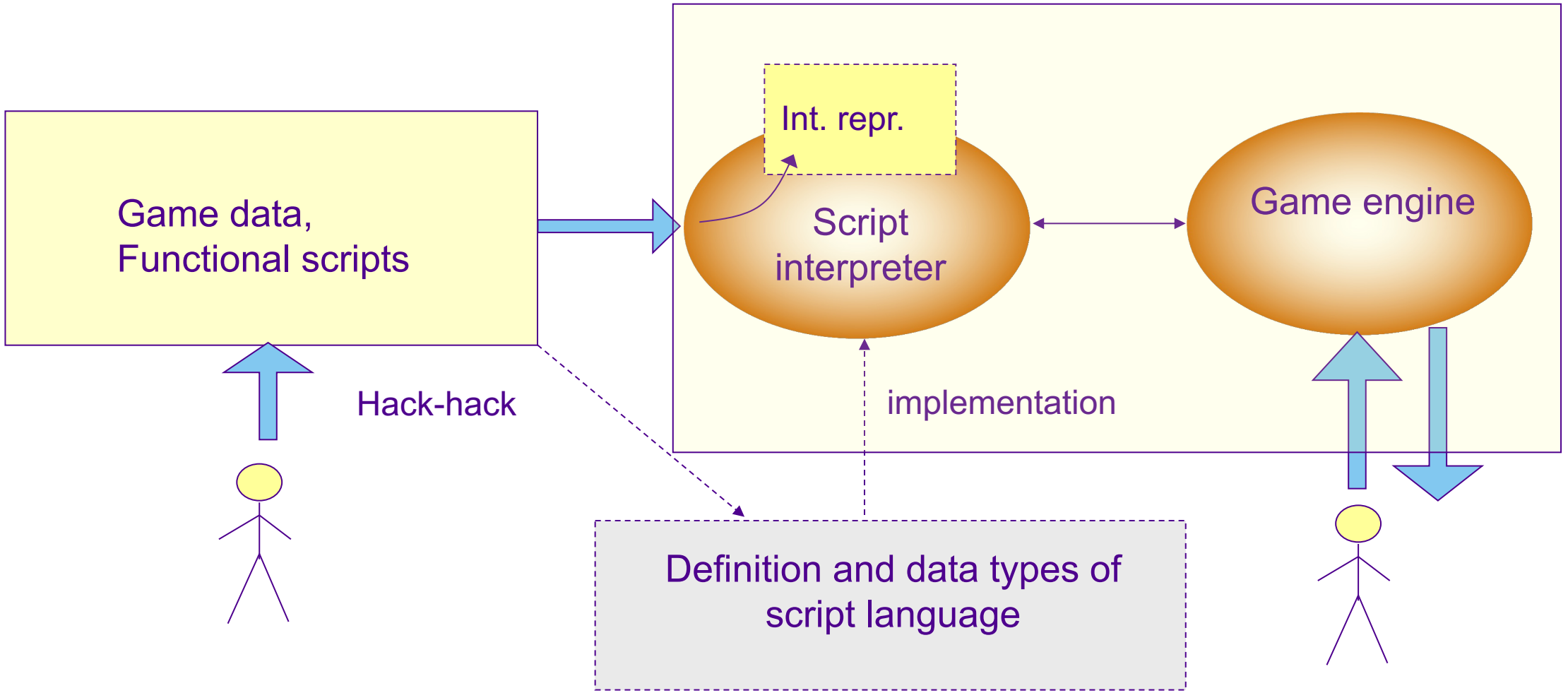
Example: SQL


DBMS
SQL inquiry
SQL interpreter
Database
Results
Application

Example: Interpreted DSL (Domain specific language)



Example: Modifiable games



Modifying games and extensions

- Skyrim: http://www.creationkit.com/Main_Page
  - Tools for creating maps, persons, stories, etc.
  - Papyrus scripts
  - Also Fallout 3 / New Vegas, Oblivion
- Medieval Total War 2: http://medieval2.heavengames.com/m2tw/mod_portal/tutorials/index.shtml
  - Definition of troops and building by text files.
- Civilization 5: http://modiki.civfanatics.com/index.php/Main_Page
  - XML, Lua scripts

Notes

- Defining an own language and implementing an interpreter system is not always the most sensible alternative. Always check if any existing one would do.

Pros and cons of interpreter architectural style

- Pros
  - Run-time logical execution environment in own control
  - The language to be interpreted relatively easy to change
  - The semantics of the language relatively easy to change
  - Underlying environment can be easily changed
- Cons
  - Performance (indirect, not-native executions; forming internal representation)
  - Memory usage (internal representation may require a lot of room)
  - Implementation and designing of interpreter part: laborous, demanding (forecasting the future)

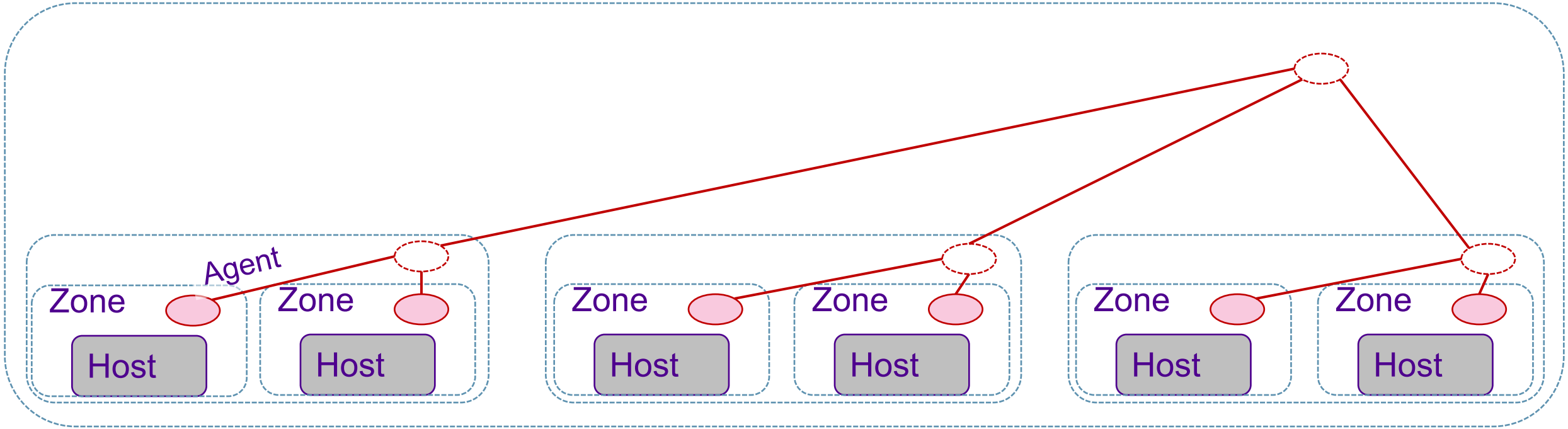
Middleware and architecture styles

- Many, such as Corba and Java RMI, are based on the object-oriented architecture.
- There are also event-based
  - TIB (The Information Bus) / Rendezvous
  - DBUS in Linux.
- JXTA is one of the peer to peer intermediate layers.

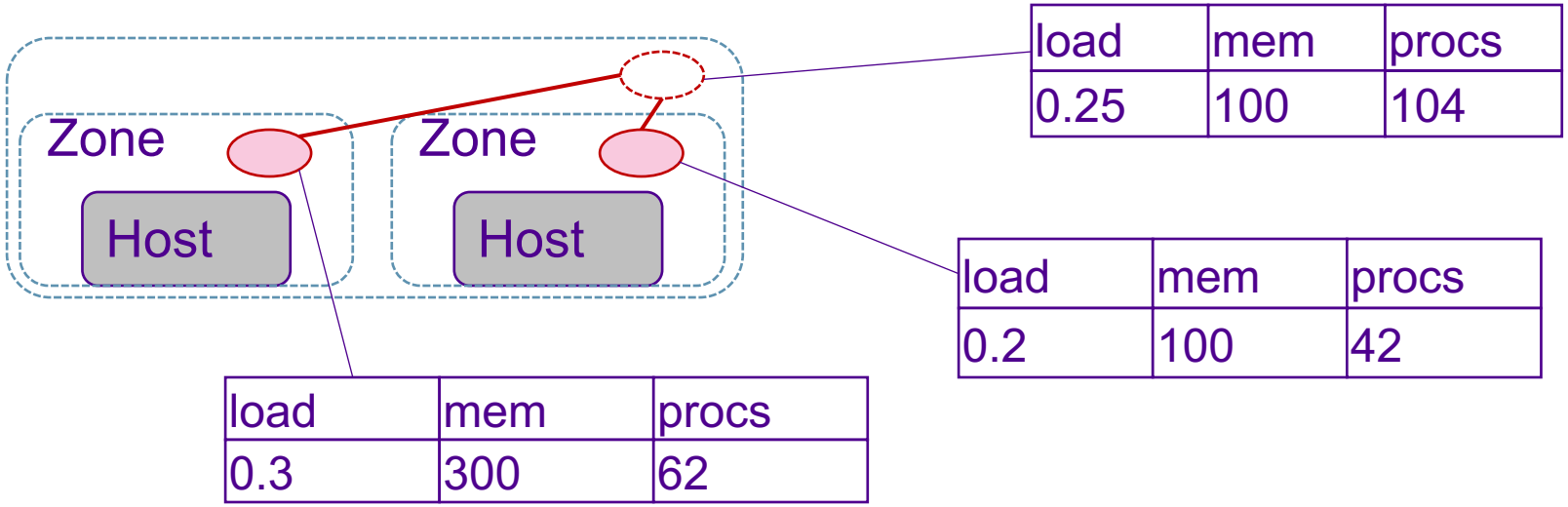
”Self-managing systems”

- Usually based on some kind of feedback loop
- Three examples
  - Astrolabe: monitor system
  - Globule: Content delivery network (CDN)
  - Jade: automatic replacement of faulty components

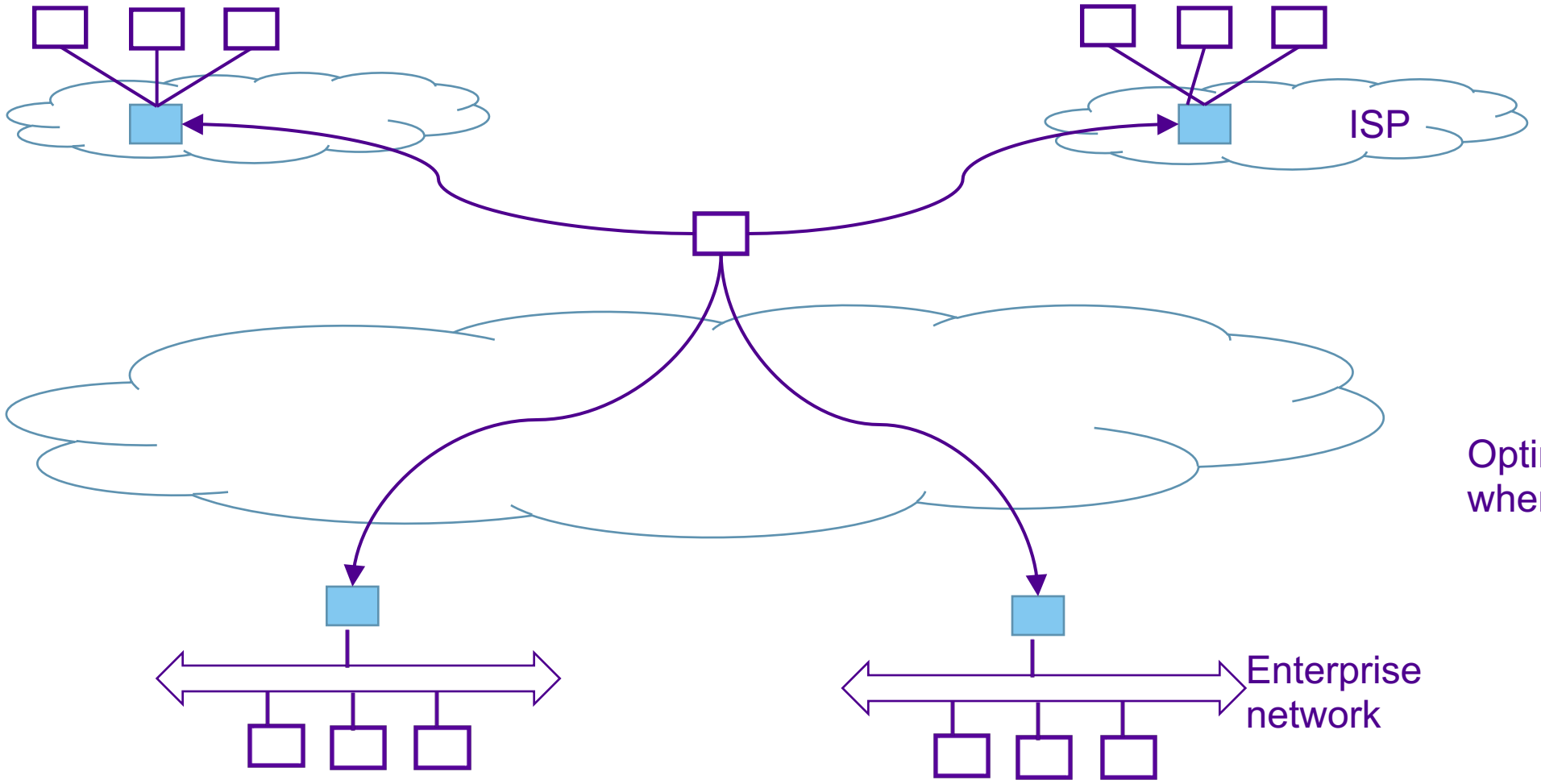
Astrolabe: Zones and Agents



Astrolabe: aggregating data

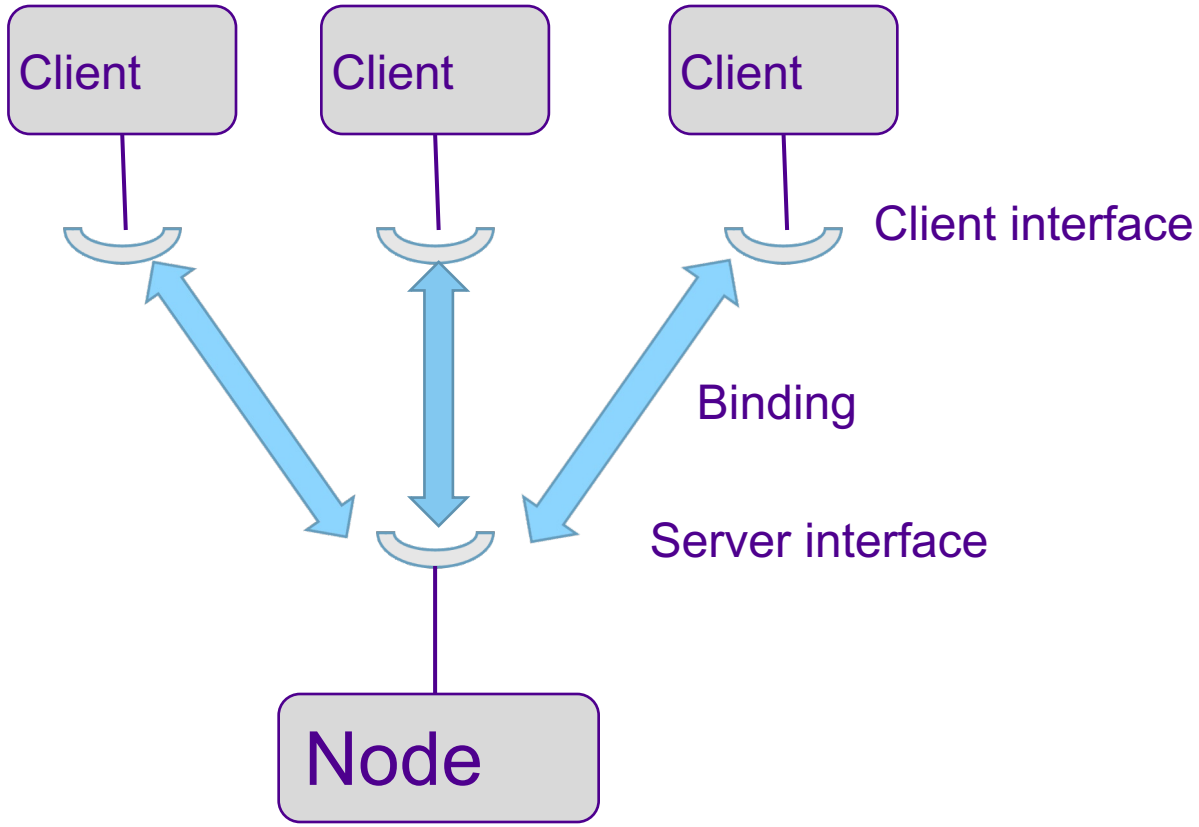


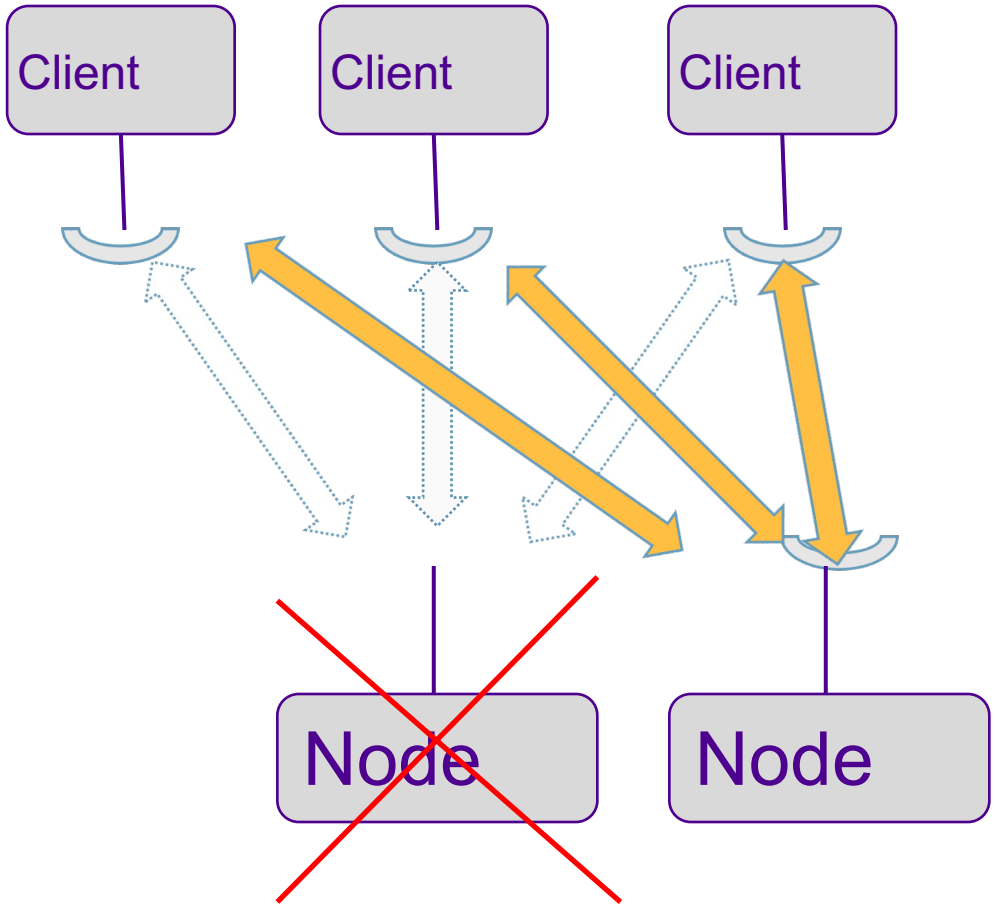
Globule: Data closer to the user
Example of CDN



Optimisation problem:
when to transfer?

Jade: automatic replacement of faulty components
Fractal componet system





Conclusions

- Architecture – Domain know-how + technical know-how
- Domain know-how ~ domain model
- Technical know-how ~ architectural styles, patterns, general good practices