Distributed objects

Agenda

Daemons
Distributed objects
    Case study: CORBA
From objects to components
    Case study: Enterprise JavaBeans
From components to frameworks
    Case study: Apache Thrift, Apache Avro
    Case study: Distributed Application Runtime (DAPR)

Questions

- Which abstractions help us to design, build and manage distributed software?

- Processes, daemons, objects, components, services, frameworks, containers, packages
  – which are the differences?

Some requirements for … what? A daemon? really?

- You write a server and you want to have it running all the time, so that clients can connect and use its service at any time. Examples: login server, web server, ftp server, etc
- You do not want your server to be killed when you logout or press ^C
- You want your server to be running in the background silently, without disturbing other daemons or getting disturbed by other daemons
- You want to run it for a long time without any manual intervention
- You want it to start automatically when the system boots up

You need a daemon!

A **daemon** is a process that:

- Starts – usually - at boot time
- Runs in the background
- It is not interactive, it is not associated with a terminal
- Terminal generated signals are not received

- **Remark**: At any time there are hundreds daemon processes running silently in any Unix/Linux/Mach system

Unix daemons

Unix-like operating systems include many daemon processes, because most services run as daemons

Common daemons:

- Web server (httpd)
- Mail server (sendmail)
- Superserver (inetd)
- System logging (syslogd)
- Print server (lpd)
- Router process (router, gated)

Example: on my Mac

```
tristano:~ paolo$ ps -uroot
  UID   PID TTY           TIME CMD
    0     1 ??        21:59.84 /sbin/launchd
    0    91 ??         2:58.41 /usr/libexec/logd
    0    92 ??         1:53.90 /usr/libexec/UserEventAgent (System)
    0    95 ??         0:07.91 /System/Library/PrivateFrameworks/Uninstall.framework/Resources/uninstalld
    0    96 ??         3:39.25 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/FSEvents.framework/Versions/A/Support/fseventsd
    0    97 ??         0:02.67 /System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremoted
    0    99 ??         1:11.20 /usr/sbin/systemstats --daemon
    0   101 ??         0:08.58 /usr/libexec/configd
    0   102 ??         0:00.01 endpointsecurityd
    0   103 ??         0:39.06 /System/Library/CoreServices/powerd.bundle/powerd
    0   108 ??         0:01.51 /usr/libexec/remoted
    0   110 ??         0:21.49 /usr/libexec/keybagd -t 15
    0   113 ??         0:01.60 /usr/libexec/watchdogd
    0   117 ??        15:45.35 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds
    0   119 ??         0:05.08 /usr/libexec/kernelmanagerd
    0   120 ??         0:10.63 /usr/libexec/diskarbitrationd
    0   124 ??         0:38.43 /usr/libexec/coreduetd
    0   125 ??         0:10.51 /usr/sbin/syslogd
    0   128 ??         0:00.03 /usr/libexec/thermalmonitord
    0   129 ??         1:16.01 /usr/libexec/opendirectoryd
    0   130 ??         0:18.66 /System/Library/PrivateFrameworks/ApplePushService.framework/apsd
    0   131 ??         0:35.12 /System/Library/CoreServices/launchservicesd
    0   136 ??         0:07.71 /usr/sbin/securityd -i
    0   137 ??         0:00.01 auditd -l
    0   141 ??         0:00.02 autofsd
    0   143 ??         1:39.38 /usr/libexec/dasd
    0   147 ??         0:00.07 /System/Library/PrivateFrameworks/AppleCredentialManager.framework/AppleCredentialManagerDaemon
    0   150 ??         0:00.05 /System/Library/CoreServices/logind
    0   151 ??         0:01.94 /System/Library/PrivateFrameworks/GenerationalStorage.framework/Versions/A/Support/revisiond
    0   152 ??         0:00.01 /usr/sbin/KernelEventAgent
    0   154 ??         2:40.27 /usr/sbin/bluetoothd
    0   155 ??         0:51.65 /usr/sbin/notifyd
    0   156 ??         0:40.72 /usr/libexec/sandboxd
```

Daemon output

No terminal, so a daemon must use something else:

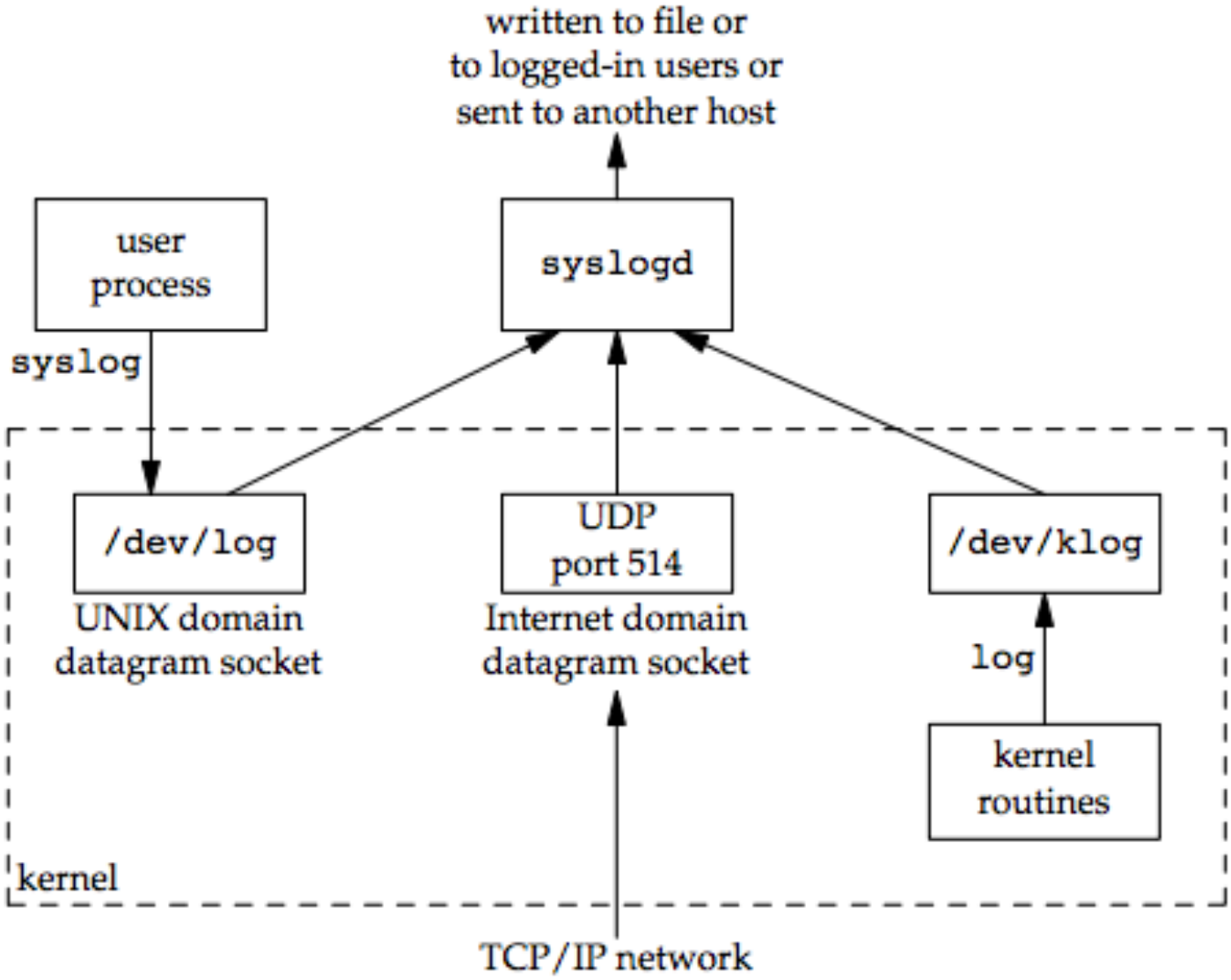
- Write some file in the file system, or
- Write in a central logging facility https://en.wikipedia.org/wiki/Syslog

Syslog is often used as logging facility: provides a central repository for system logging

The syslogd daemon provides system logging services to "clients": it is a standard for message logging

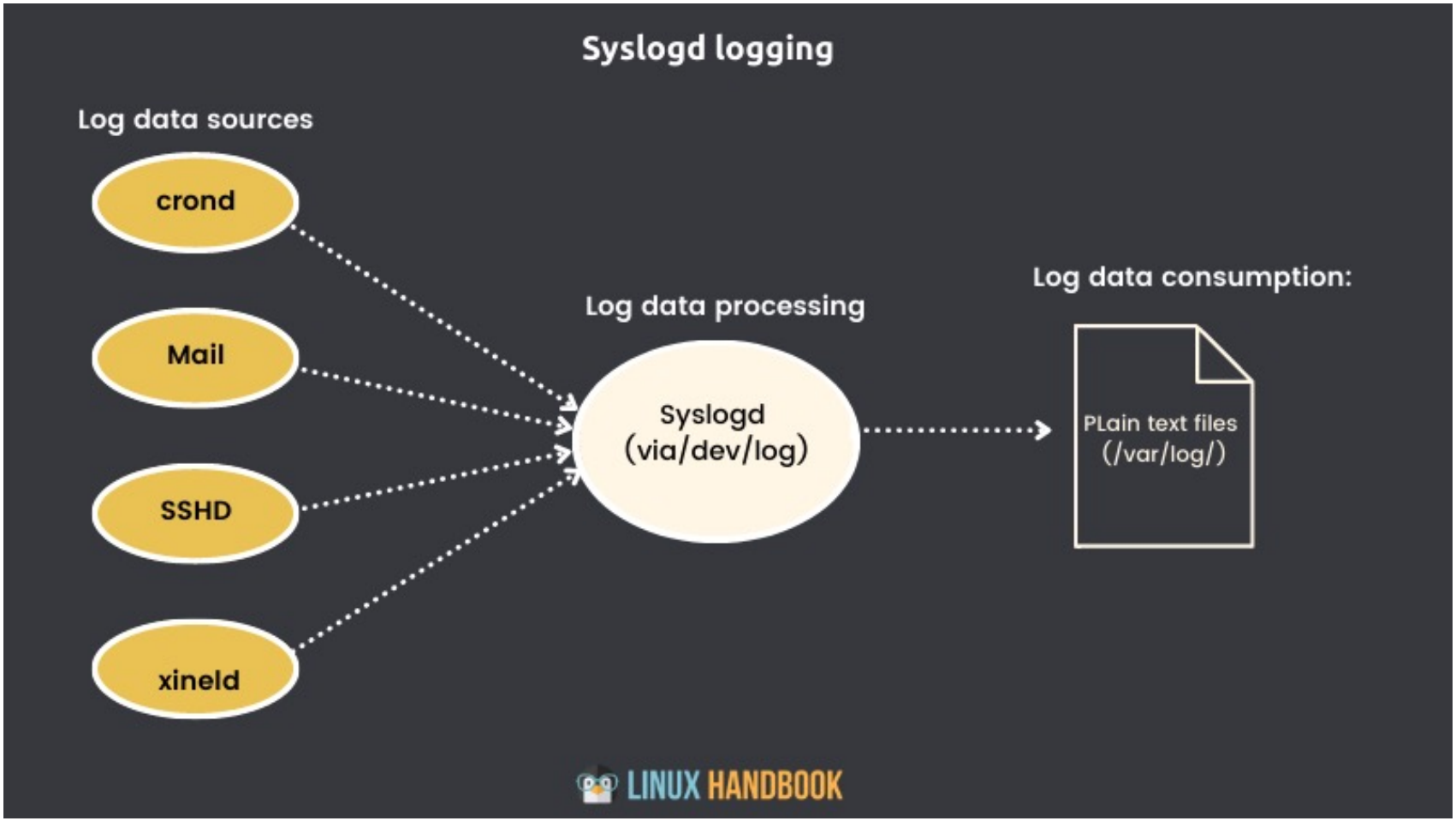
- It allows separation of the software that generates messages, the system that stores them, and the software that reports and analyzes them.
- Each message is labeled with a *facility* code, indicating the subsystem generating the message, and is also assigned a severity level (eg «*panic*» or «*device error*»)

syslogd



https://notes.shichao.io/apue/ch13/

syslogd



https://linuxhandbook.com/syslog-guide/

Exercise : Creating a Simple Linux Daemon in Python

Scenario: You are tasked with creating a Linux daemon that monitors a specific directory for new files and performs a predefined action when a new file is added to that directory. In this exercise, we'll use Python to create the daemon.

Requirements:

1. Create a Python script that will serve as the daemon process. This script should include the following functionalities:
    1. Monitor a specific directory (you can define the directory path) for new files.
    2. When a new file is added to the directory, print a log message indicating the file's name and the time it was added.
2. Implement the daemonization process to detach the script from the terminal and run it as a background daemon.
   You can use the `python-daemon` library for this purpose.
3. Configure the daemon to run indefinitely, periodically checking for new files in the monitored directory.
4. Show a test with a log of changes in two different days – possibly automate the test using the `cron` command

Daemons, services, and objects

A daemon is a Linux process existing at run-time and offering specific services to users and administrators

An object is a component of an object-oriented system, it offers services to other objects in the system

While a daemon is a process, so it only needs to be started inside an operating system, an object (eg. written in Java, or in Python) can be local or remote, and needs middleware to be useful

Middleware frameworks for distributed objects/components

- A complete middleware platform must offer high-level programming abstractions as well as making transparent the underlying complexities involved in distributed systems
- We examine here two of the most important programming abstractions, namely distributed objects and components, and some related middleware platforms including CORBA, Enterprise JavaBeans, Thrift, DAPR

Before Distributed Objects

- In the early 1990s desktops were used to display and edit data provided by mainframes and minicomputers.
- For instance, SQL required the workstation to download huge data sets and then process them locally, whereas use of terminal emulators left all of the work to the server and provided no GUI.
- The proper split of duties would be to have a cooperative set of objects, the workstation being responsible for display and user interaction, with processing on the server.
- Standing in the way of this solution were the massive differences in operating systems and programming languages between platforms.

Before Distributed Objects - 2

- The differences between any two programming languages on a single platform were almost as great.
- Each language had its own format for passing parameters into procedure calls, the file formats that they generated were often quite different.
- The problem was not so acute on minicomputers and mainframes where the vendor often specified standards for their libraries, but on microcomputers the programming systems were generally delivered by a variety of 3rd party companies with no interest in standardization.

What is middleware

Middleware is a type of software that provides services to applications. These services extend those available from the operating system.

Middleware can be described as «software glue»

Main types of middleware:

- **Transactional**: Processing of multiple synchronous/ asynchronous requests or queries, such as bank transactions or credit card payments.
- **Message-oriented**: message passing architectures, including message queues, which support synchronous/asynchronous communication.
- **Object-oriented**: incorporates object-oriented programming design principles, like object references, exceptions, and inheritance of properties via distributed object requests.

Middleware for distributed objects: modifications to some concepts

Objects	Distributed objects	Description
Object reference	Remote object reference	Globally unique reference to a remote object; may be passed as parameter
Interface	Remote interface	Provides an abstract specification of the methods that can be invoked on the remote object; it is an API specified using an Interface Description Language (IDL)
Action	Distributed action	Remote invocations use RMI
Exception	Distributed exception	Additional exceptions generated for supporting distribution: eg. Message loss or process failure
Garbage collection	Distributed garbage collection	An object exists until a local or remote reference to it exists; otherways it should be removed. Requires a distributed garbage collection algorithm

Component based middleware

- Component-based middleware has emerged as a natural evolution of distributed objects, providing:
  - support for managing dependencies between components,
  - hiding low level details associated with the middleware,
  - managing the complexities of establishing distributed applications with appropriate non-functional properties such as security,
  - and supporting appropriate deployment strategies.
- A famous example in this area is Enterprise JavaBeans.

Objects vs components

Objects

Self contained (small)

Provide an abstraction

Represent real entities

Stateful, Operations (methods)

Useful for solving a problem

Fine-grained, clearly defined

Components

Self contained (large)

Provide services

Use stated interfaces

Stateless

Useful when reused

Large-grained, coherent

Added complexity

- Inter-object communication
- Object lifecycle management (eg versioning)
- Activation and deactivation
- Persistence
- Additional services (naming, security and transaction services)

MISSION & VISION

MISSION STATEMENT

The mission of the Object Management Group (OMG) is to develop technology standards that provide real-world value for thousands of vertical industries. OMG is dedicated to bringing together its international membership of end-users, vendors, government agencies, universities and research institutions to develop and revise these standards as technologies change throughout the years.

VISION

The Object Management Group® (OMG®) is an international, open membership, not-for-profit **technology standards** consortium, founded in 1989. OMG standards are driven by vendors, end-users, academic institutions and government agencies. OMG Task Forces develop enterprise integration standards for a wide range of technologies and an even wider range of industries. OMG's modeling standards, including the Unified Modeling Language® (UML®) and Model Driven Architecture® (MDA®), enable powerful visual design, execution and maintenance of software and other processes. OMG also hosts organizations such as the Consortium for Information & Software Quality™ (CISQ™), the DDS Foundation and BPM+ Health. In addition, OMG manages the Industrial Internet Consortium® (IIC™), the Industry IoT Consortium® and Digital Twin Consortium™.

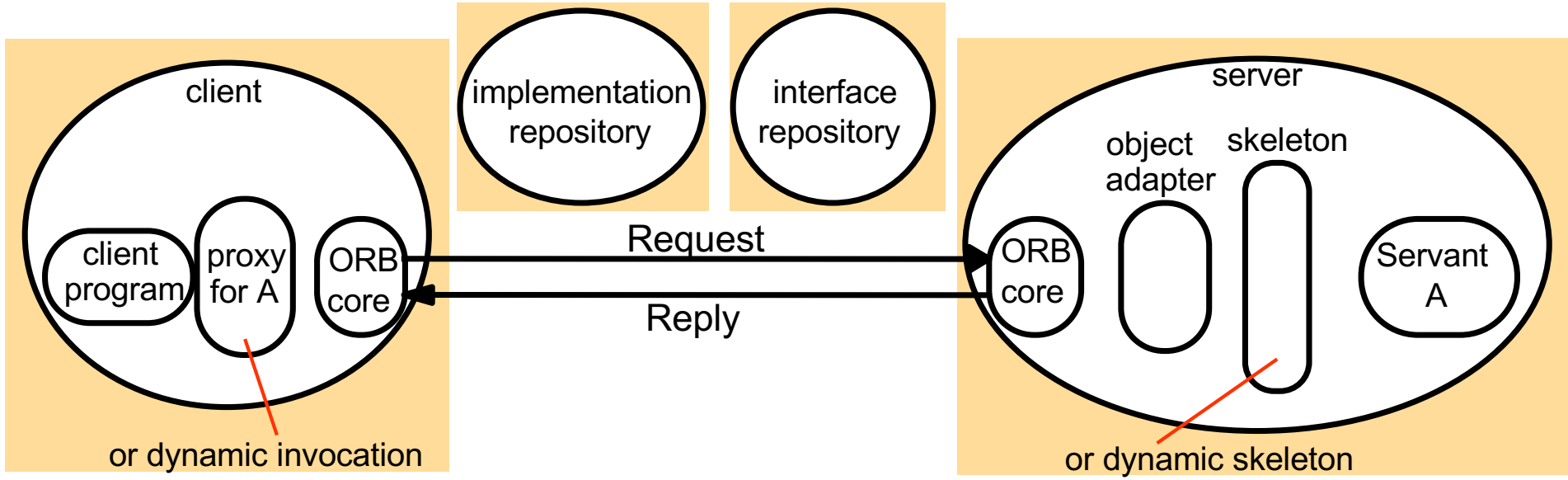
OMG and CORBA

- The Object Management Group (OMG) was formed in 1989 to foster the adoption of distributed object systems, so to gain the benefits of object-oriented programming for software development and to exploit distributed systems, which were becoming widespread
- To achieve its aims, the OMG advocated the use of open systems based on standard object-oriented interfaces.
- These systems would be built from heterogeneous hardware, networks, operating systems and programming languages.
- An important motivation was to allow distributed objects to be implemented in any programming language and to be able to communicate with one another.
- The OMG therefore designed an interface language that was independent of any specific implementation language.
- They introduced a metaphor, the object request broker (ORB), whose role is to help a client to invoke a method on an object

CORBA https://www.corba.org

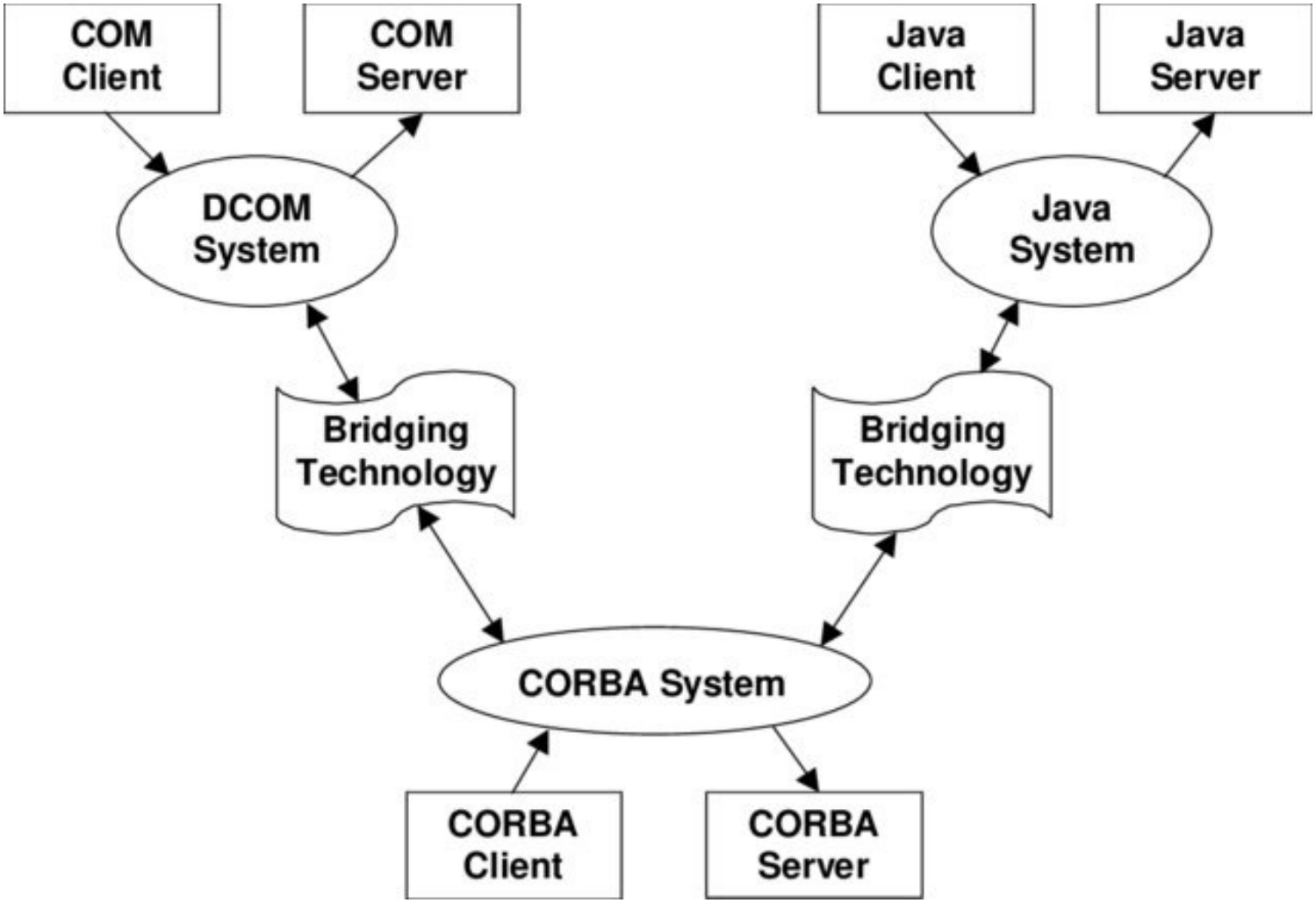
- CORBA is a reference middleware designed to allow applications to communicate with each other irrespective of programming languages, hardware, and operating systems: its main goal was **interoperability**
- Applications are built from interfaces defined in CORBA's interface definition language IDL.
- CORBA supports transparent invocation of methods on remote objects.
- The middleware component that supports remote invocations is called the object request broker, or ORB

The main components of the CORBA architecture



The CORBA architecture is designed to support the role of an object request broker that enables clients to invoke methods in remote objects, where both clients and servers can be implemented in a variety of programming languages.

CORBA as interoperability platform



Components [Szyperski 2002]

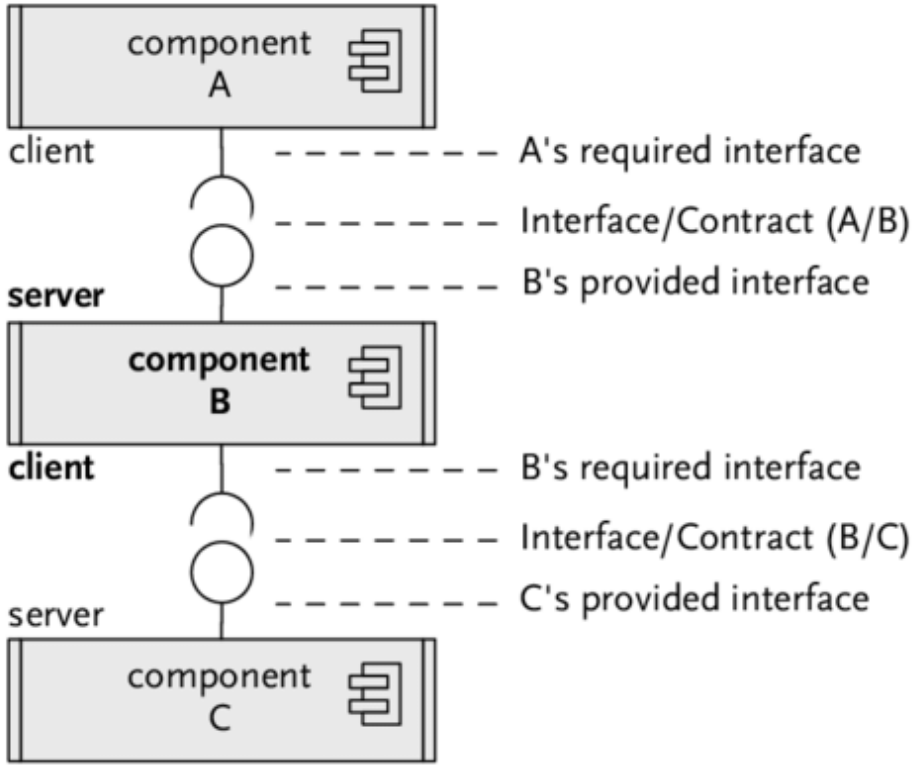
Definition: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only

- In this definition, the word ‘only’ refers to the fact that any context dependencies must be explicit – that is, there are no implicit dependencies
- Software components are like distributed objects in that they are encapsulated units of composition, but a given component specifies both its interfaces provided to the outside world and its dependencies from other components
- The dependencies are also represented as interfaces.

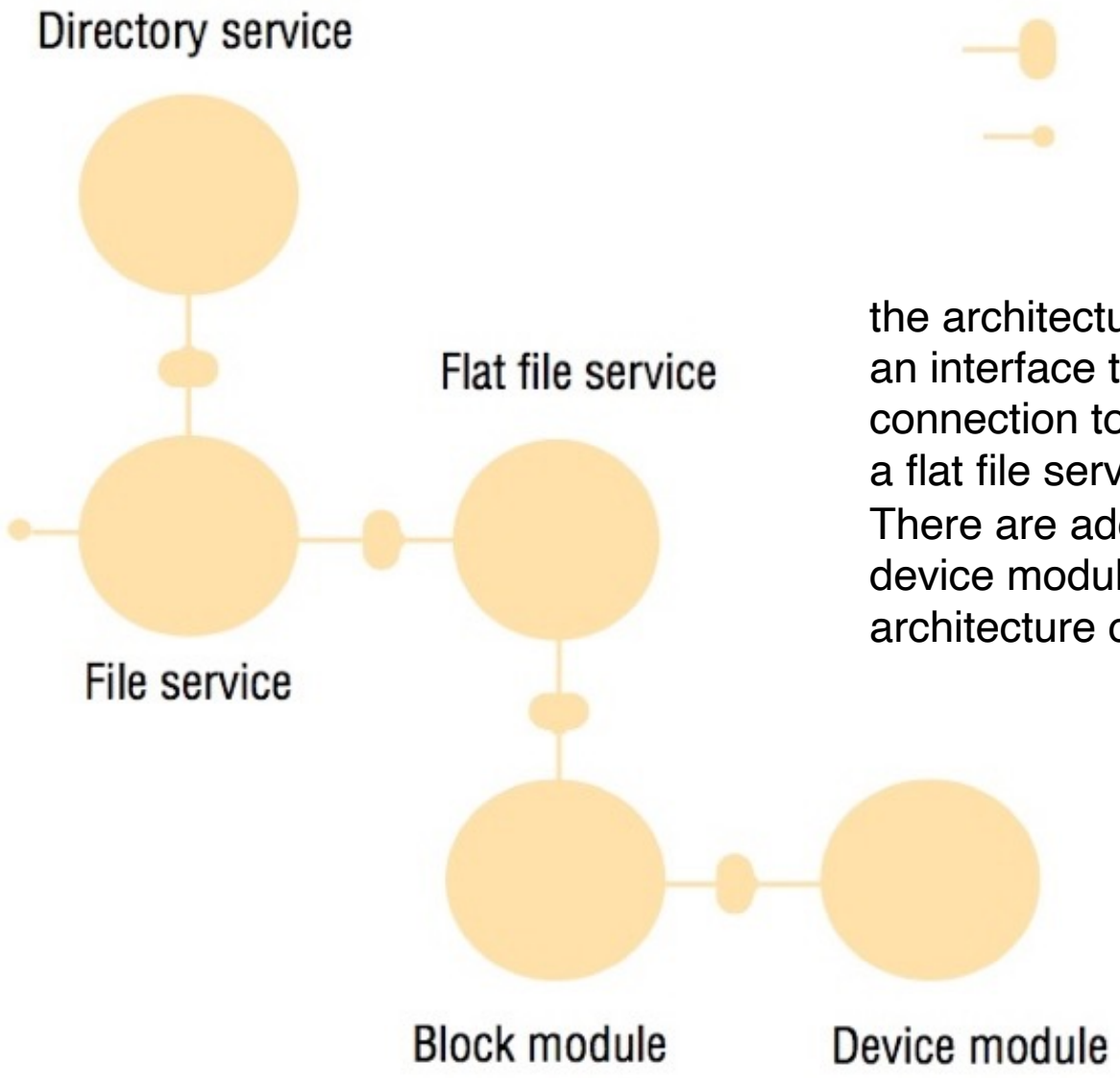
Components have contracts

A component is usually specified in terms of a *contract*, which includes:

- a set of *provided* interfaces – that the component offers as services to other components;
- a set of *required* interfaces – that are the dependencies from other components that must be present and connected to this component for it to function correctly



An example software architecture including components

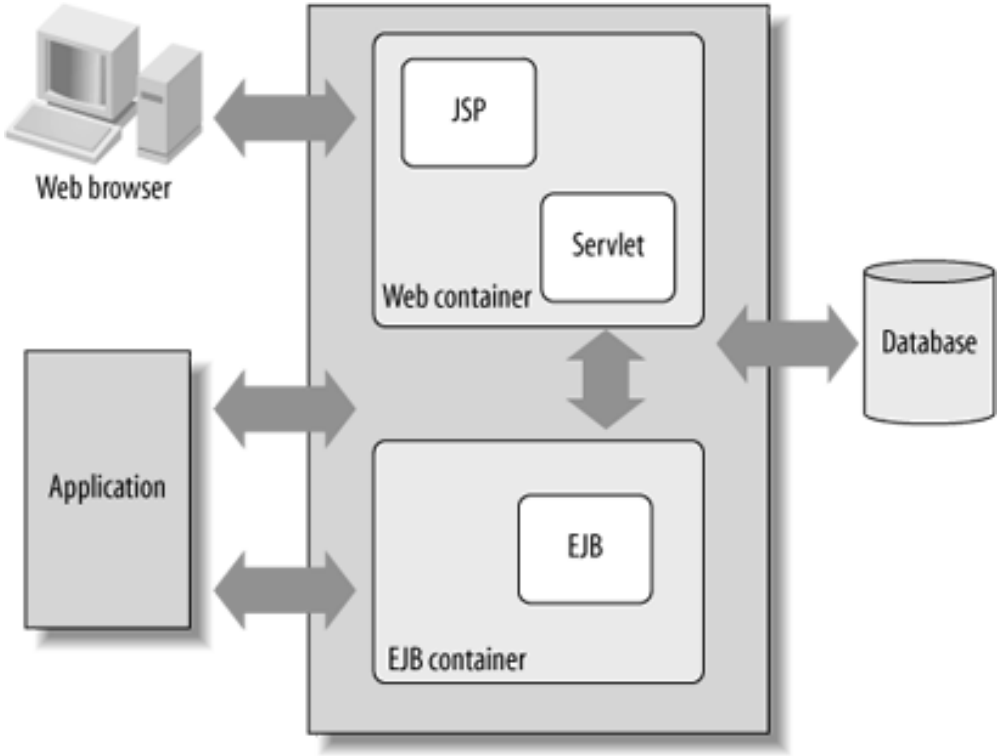


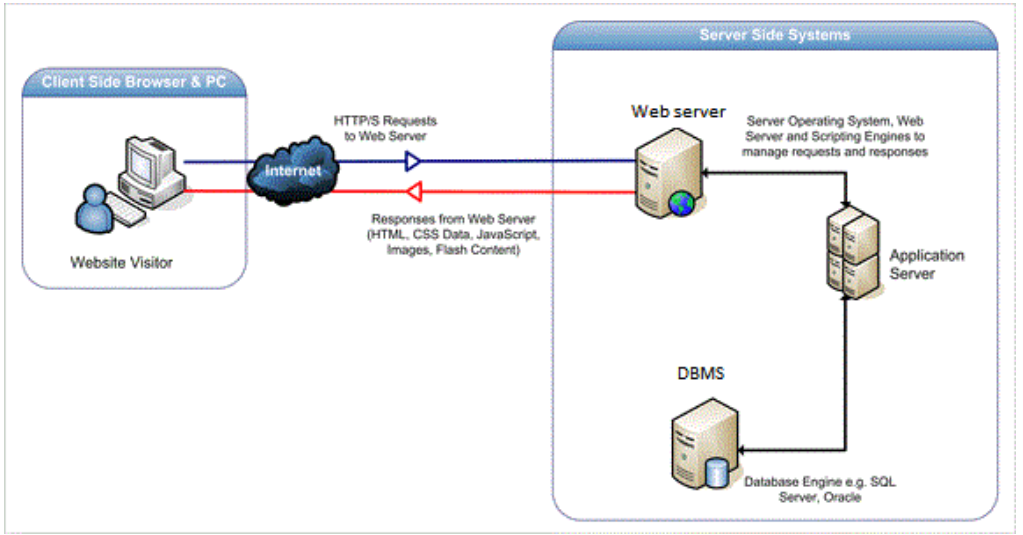
the architecture of a simple file system providing an interface to other users and in turn requiring connection to a directory service component and a flat file service component.
There are additional connections to block and device modules, capturing the overall architecture of this particular file system

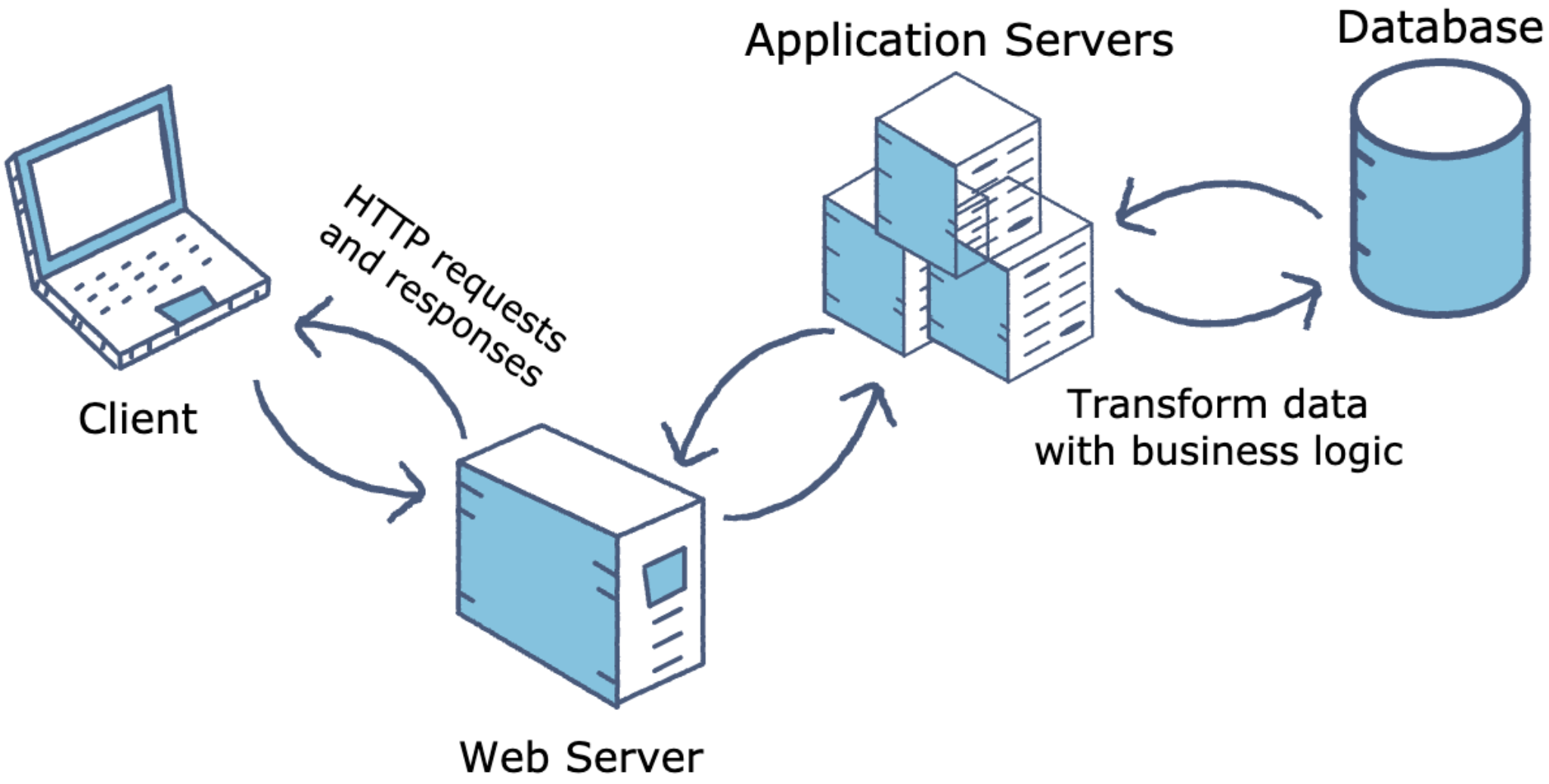
Web servers and application servers

Web Server and Application server are often used together. However:

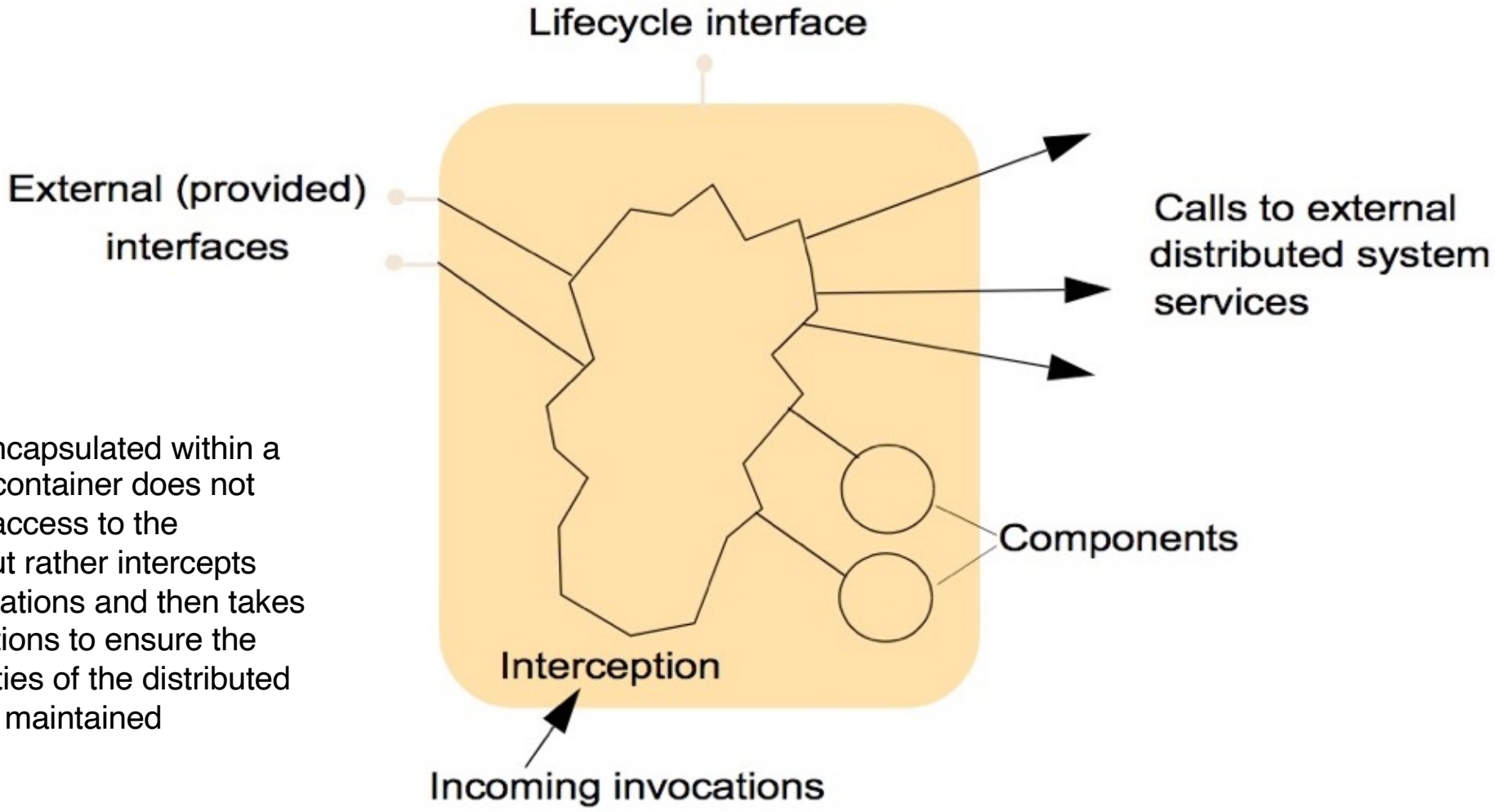
- Web Server is designed to serve HTTP Content. App Server is not limited to just HTTP. It can provide other protocol support such as RMI/RPC
- Most application servers have a Web Server as integral part of them. Additionally, App Server have components and features to support application level services such as Connection Pooling, Object Pooling, Transaction Support, Messaging services etc.
- As Web servers are well suited for static content and app servers for dynamic content, most of the production environments have web server acting as reverse proxy to app server. That means while servicing a page request, static contents (such as images/Static HTML) are served by web server that interprets the request.
  Example: Tomcat HTTP server






Application Servers
Database
HTTP requests
and responses
Client
Transform data
with business logic
Web Server

The structure of an application server



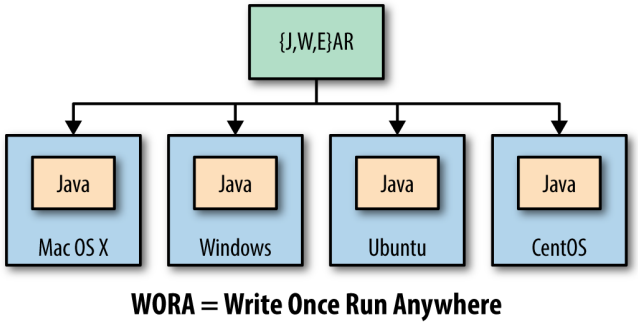
components encapsulated within a container; the container does not provide direct access to the components but rather intercepts incoming invocations and then takes appropriate actions to ensure the desired properties of the distributed application are maintained

Application servers

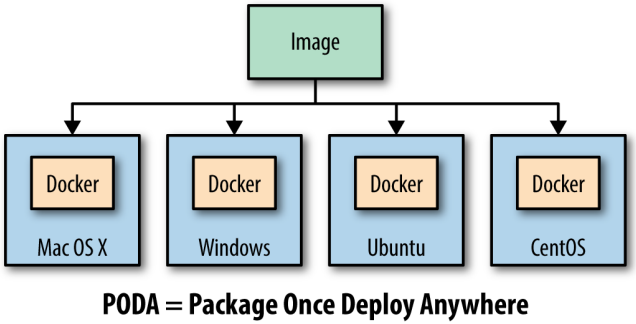
Product	Vendor	Last release	License
WildFly/Jboss	RedHat	2022	LGPL
ColdFusion	Adobe	2016	Proprietary
Geronimo	ASF	2013	Apache
GlassFish	Eclipse	2022	GPL
Tomcat	ASF	2022	Apache
WebSphere	IBM	2016	Proprietary
Jetty	Eclipse	2022	Apache

https://en.wikipedia.org/wiki/List_of_application_servers

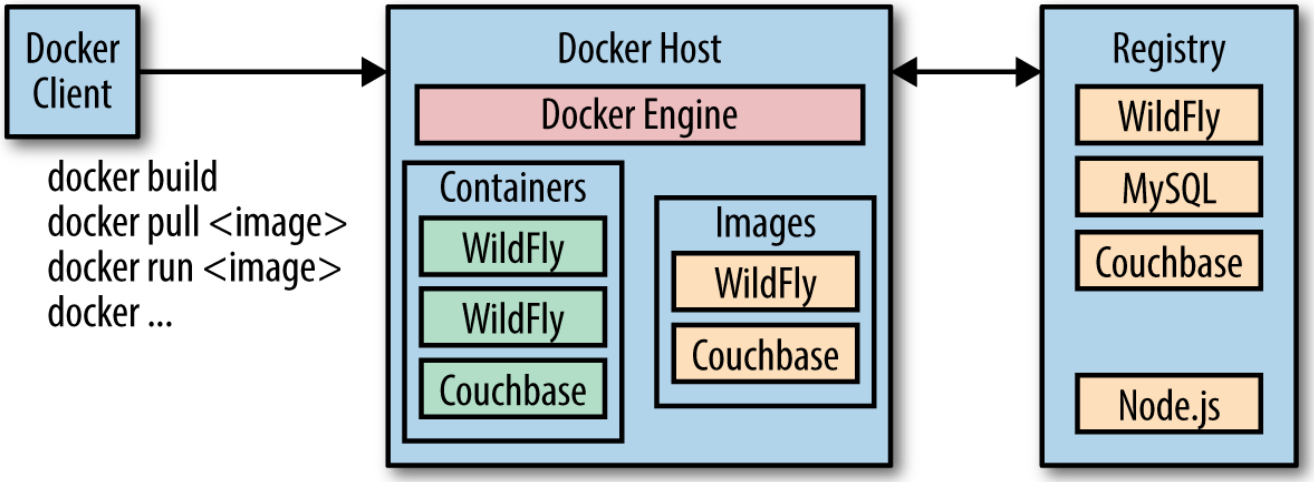
Java, WildFly, Docker



WORA = Write Once Run Anywhere



PODA = Package Once Deploy Anywhere



https://www.oreilly.com/library/view/docker-for-java/9781492042624/ch01.html

A typical developer workflow involves running Docker Engine on a host machine. It does the heavy lifting of building images, and runs, distributes, and scales Docker containers. The client is a Docker binary that accepts commands from the user and communicates back and forth with the Docker Engine. Couchbase is a NoSQL cloud database service

Enterprise Java Beans (EJB)

- The EJB specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems
- The EJB specification provides a standard way to implement the server-side (also called '*back-end*') 'business' software found in enterprise applications (as opposed to '*front-end*' user interface software).
- Such code often addresses well known problems, and solutions to these problems are often repeatedly re-implemented by programmers.
- Enterprise JavaBeans is intended to handle such common concerns as: persistence, transactional integrity, and security in a standard way, leaving programmers free to focus on the particular parts of the bjsiness logic

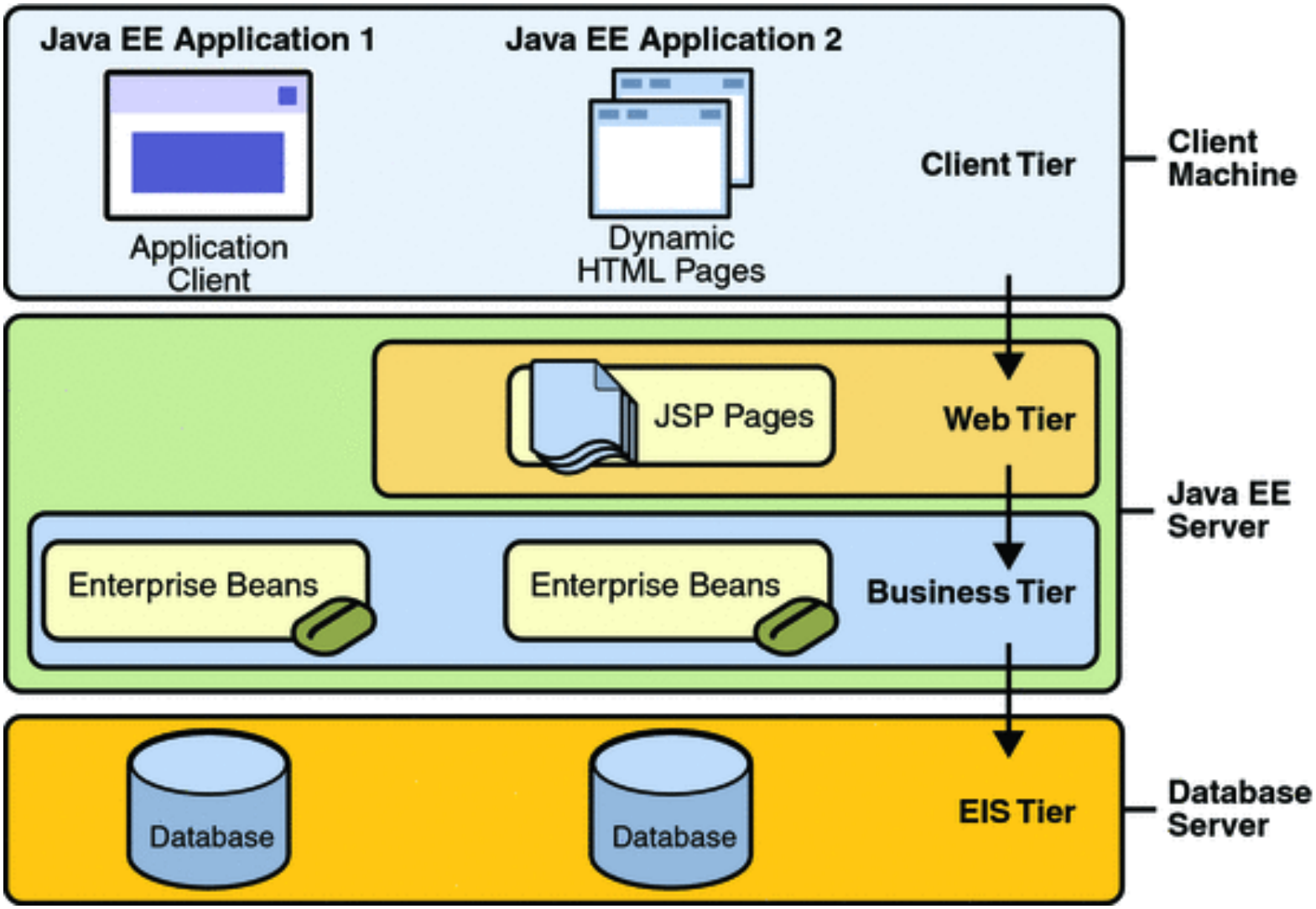
EJB

- EJB is a server-side software component that encapsulates the business logic of an application.
- An EJB web container provides a runtime environment for several software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services.
- The EJB specification is a subset of the JavaEE specification

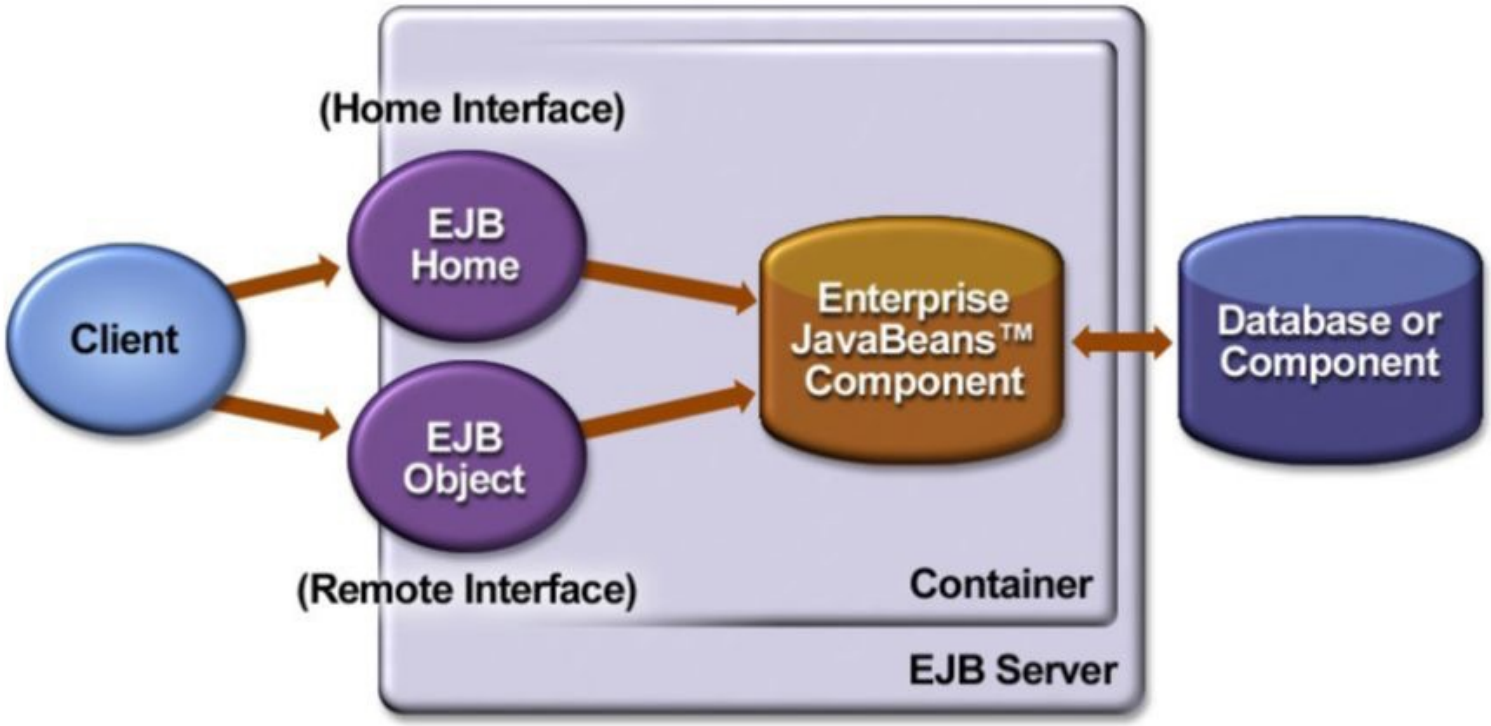
Java EE (Enterprise Edition, Jakarta)

- Java EE is defined by its specification.
- The specification defines APIs and their interactions.
- As with other Java Community Process specifications, providers must meet certain conformance requirements in order to declare their products as Java EE compliant.
- Java EE applications are run on reference runtimes, that can be microservices or application servers, which handle transactions, security, scalability, concurrency and management of the components it is deploying.

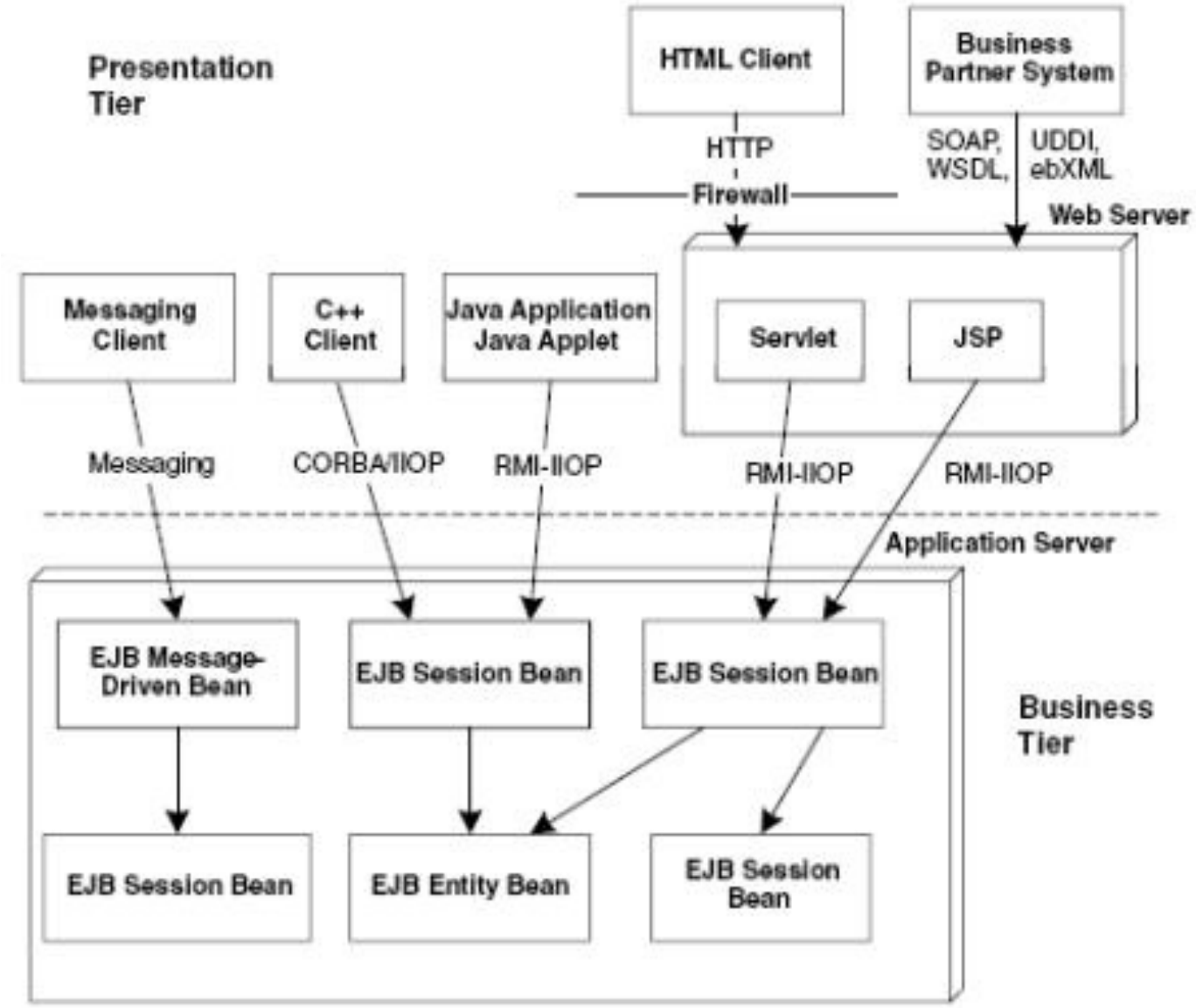
Java EE stack



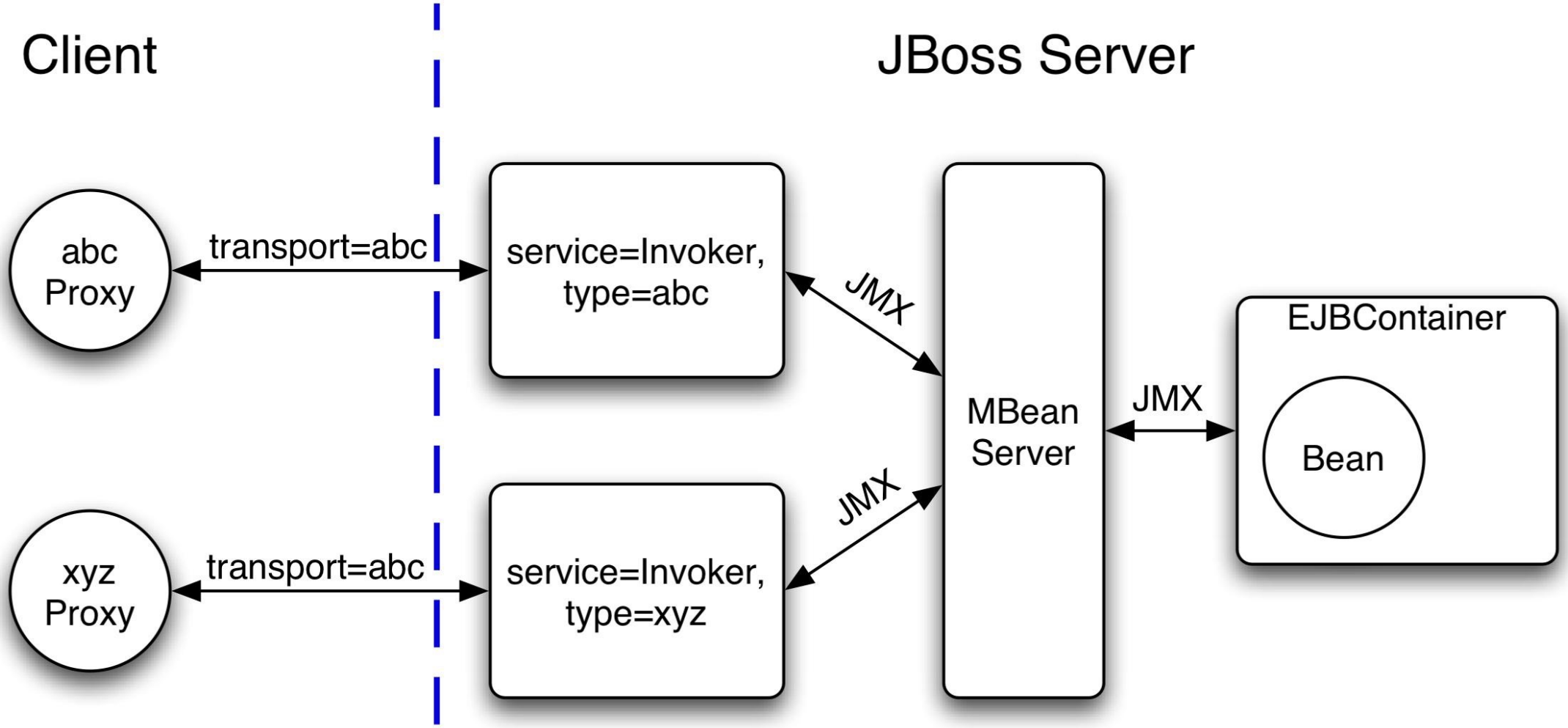
EJB Architecture



EJB in a three tier architecture



JBoss (Wildfly) server

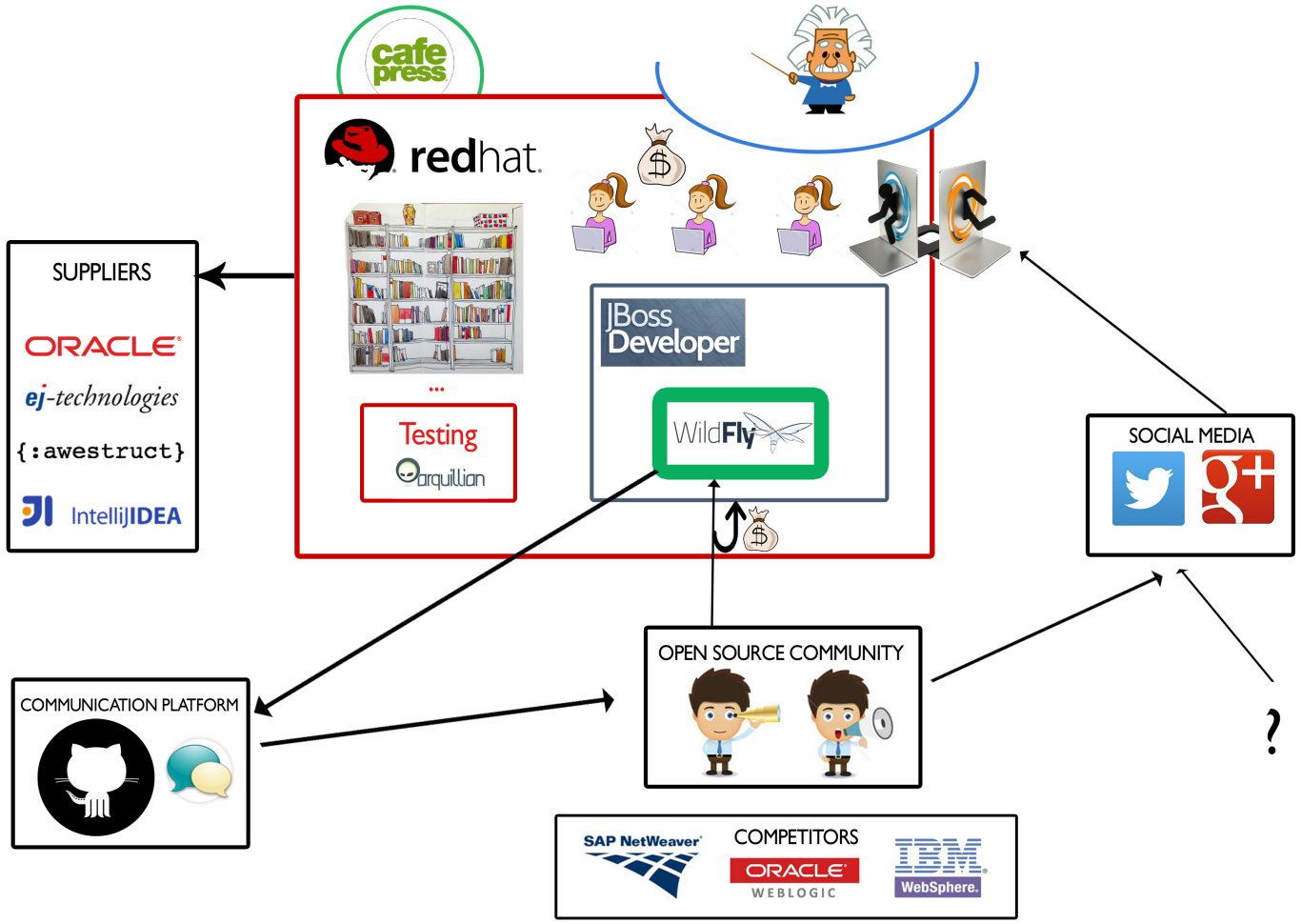


Wildfly context diagram

WildFly is a server application that provides easy communication between the user and its server. It is one of the many features of the JBoss Application Enterprise.

Owned by RedHat

WildFly provides remote management for one or more servers and runs on a local host server as well

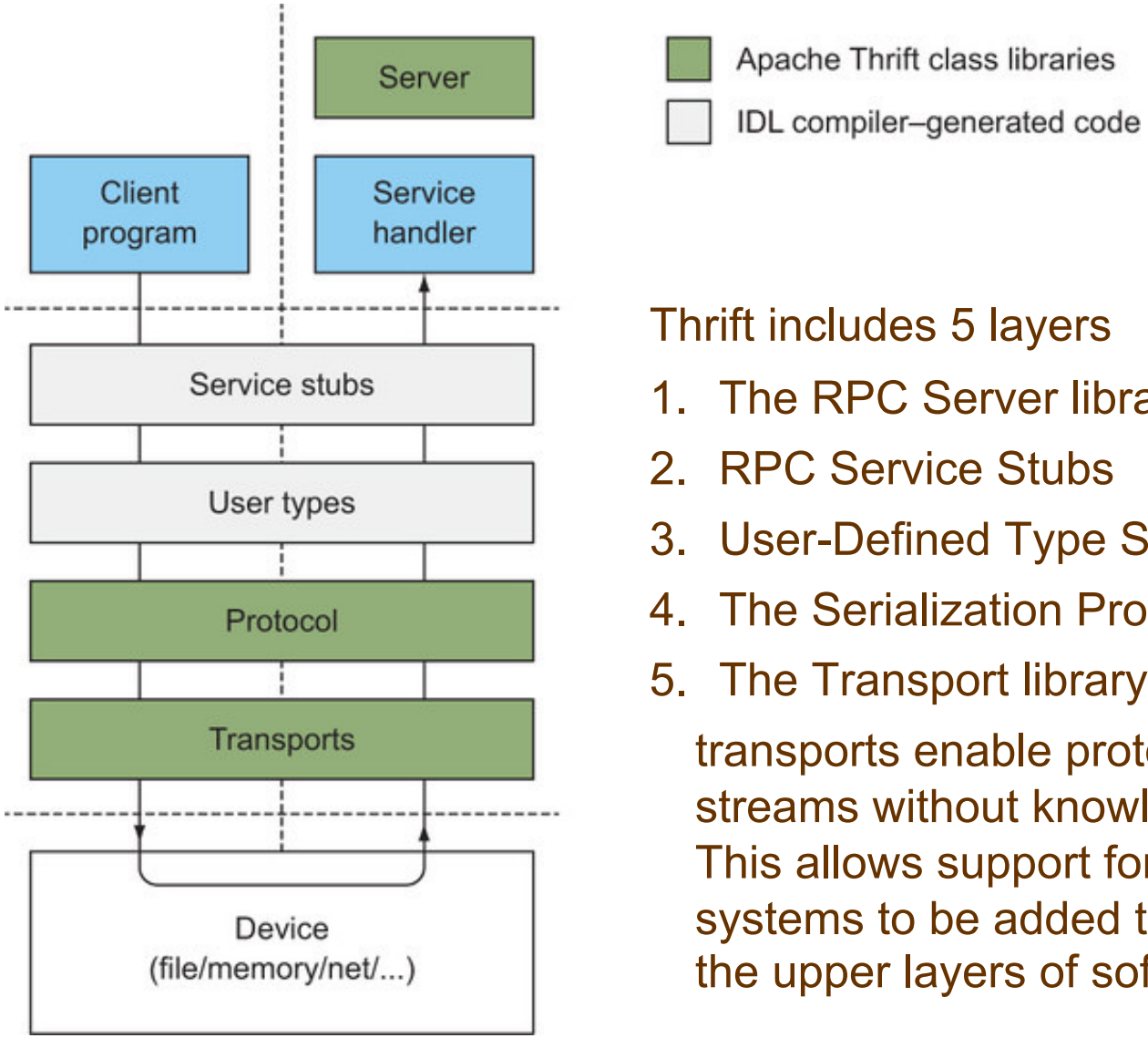


https://delftswa.gitbooks.io/desosa2016/content/wildfly/Chapter.html

Apache Thrift

- Thrift is a framework including an IDL and a binary communication protocol used for defining and creating services for several languages.
- It forms a remote procedure call (RPC) framework
- It was developed at Facebook for "scalable cross-language services development".
- It combines a software stack with a code generation engine to build cross-platform services which can connect applications written in a variety of languages and frameworks,
- Thrift includes a complete stack for creating clients and servers.

Apache Thrift



Thrift includes 5 layers

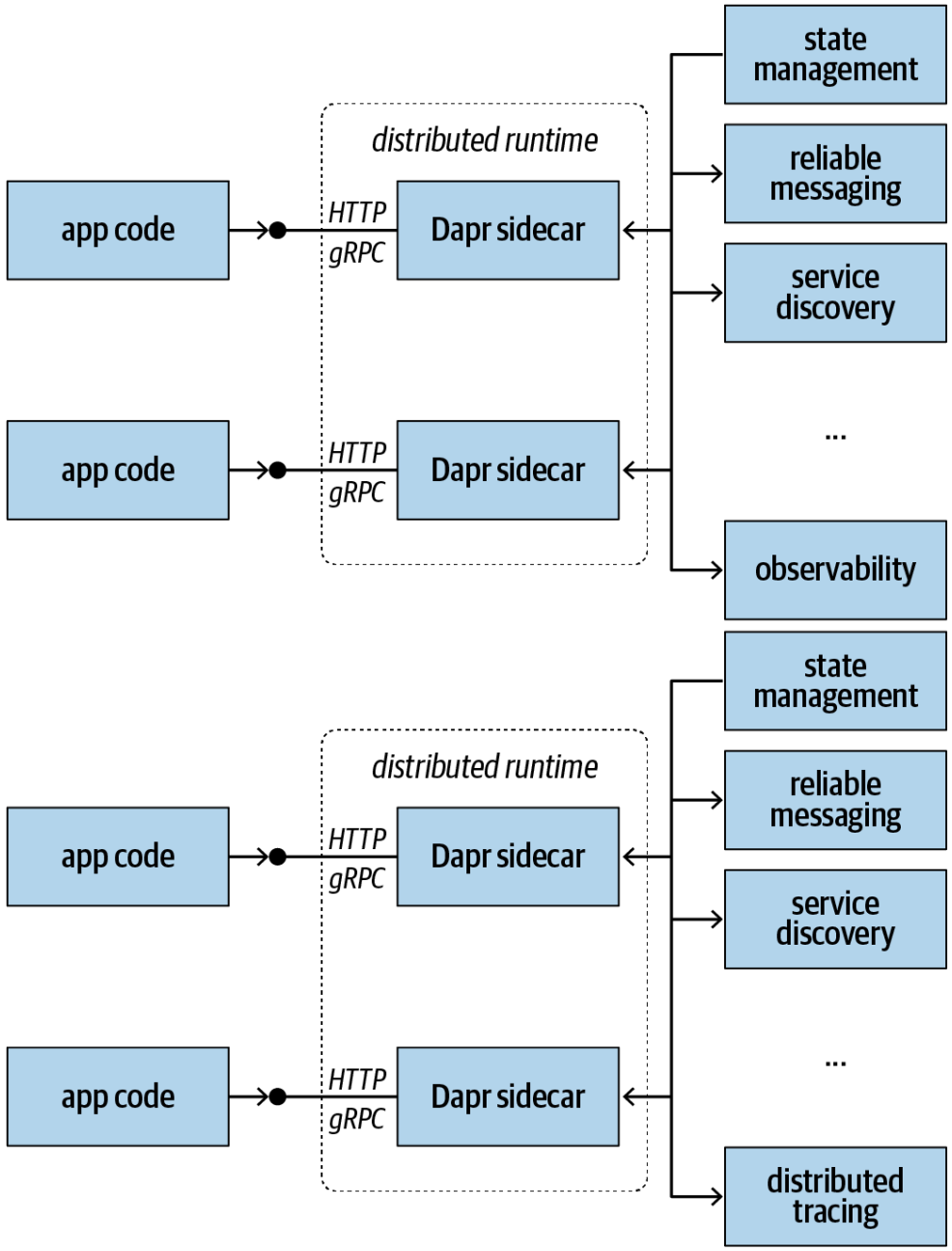
1. The RPC Server library
2. RPC Service Stubs
3. User-Defined Type Serialization
4. The Serialization Protocol library
5. The Transport library

transports enable protocols to read and write byte streams without knowledge of the underlying device. This allows support for new devices and middleware systems to be added to the platform without impacting the upper layers of software

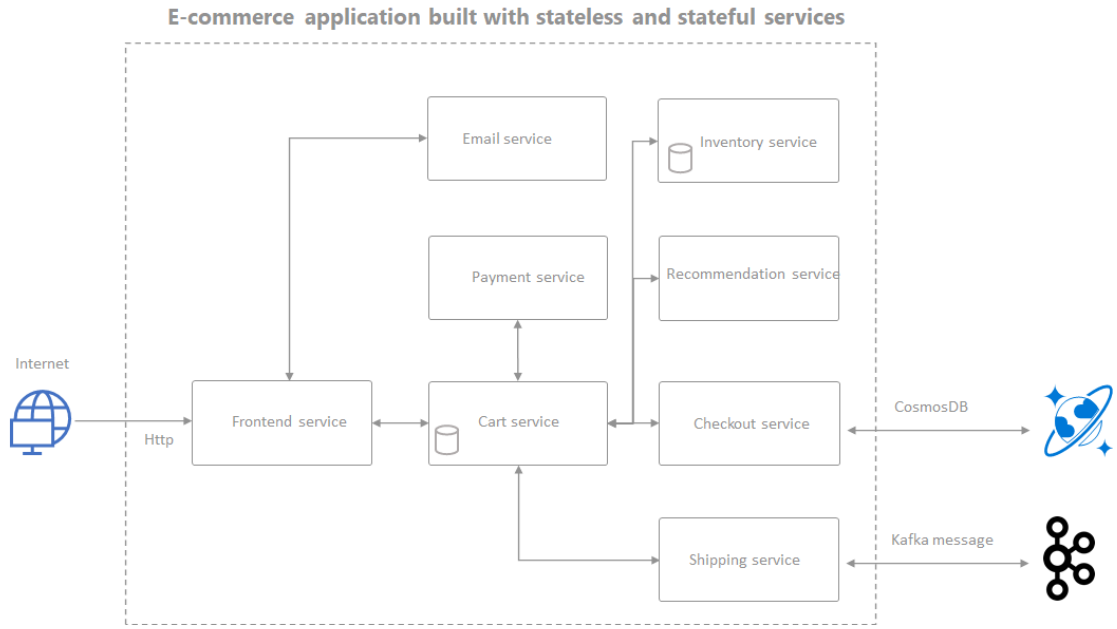
Distributed Application Runtime DAPR dapr.io

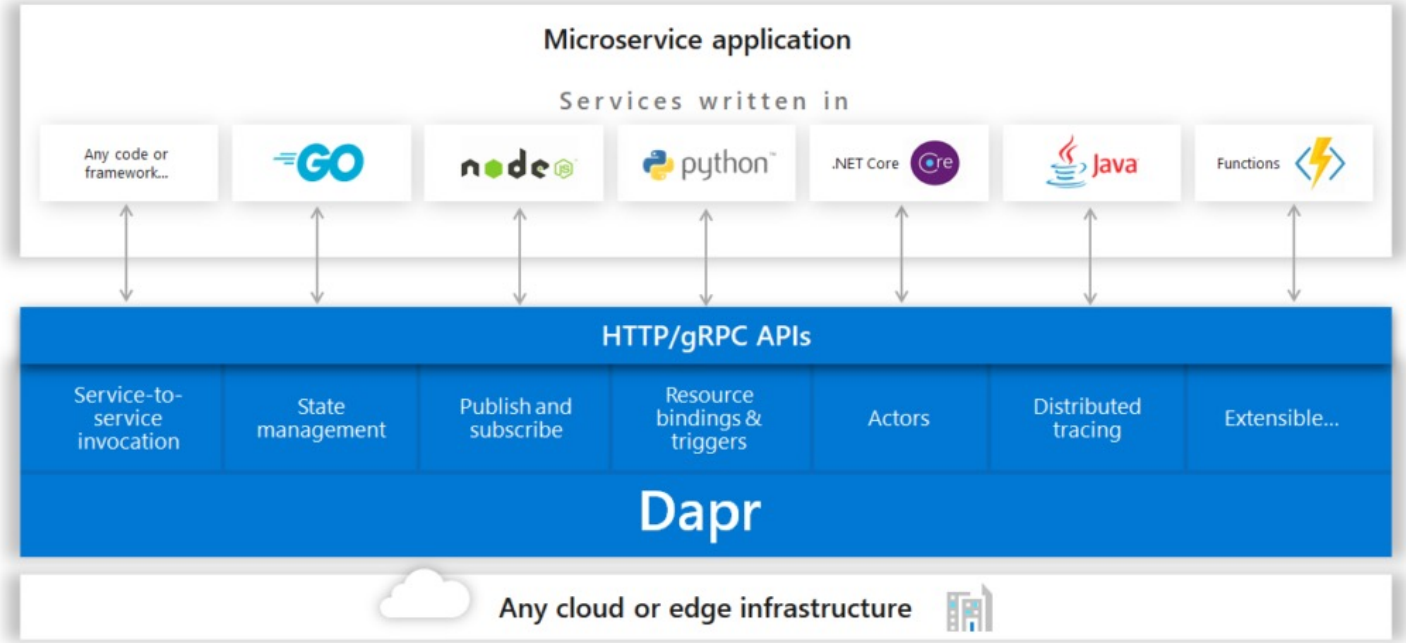
DAPR is a an event-driven, portable runtime for building microservices for the cloud and the edge

DAPR offers a unified programming model that delivers capabilities through a standardized HTTP/gRPC protocol".



DAPR and gRPC

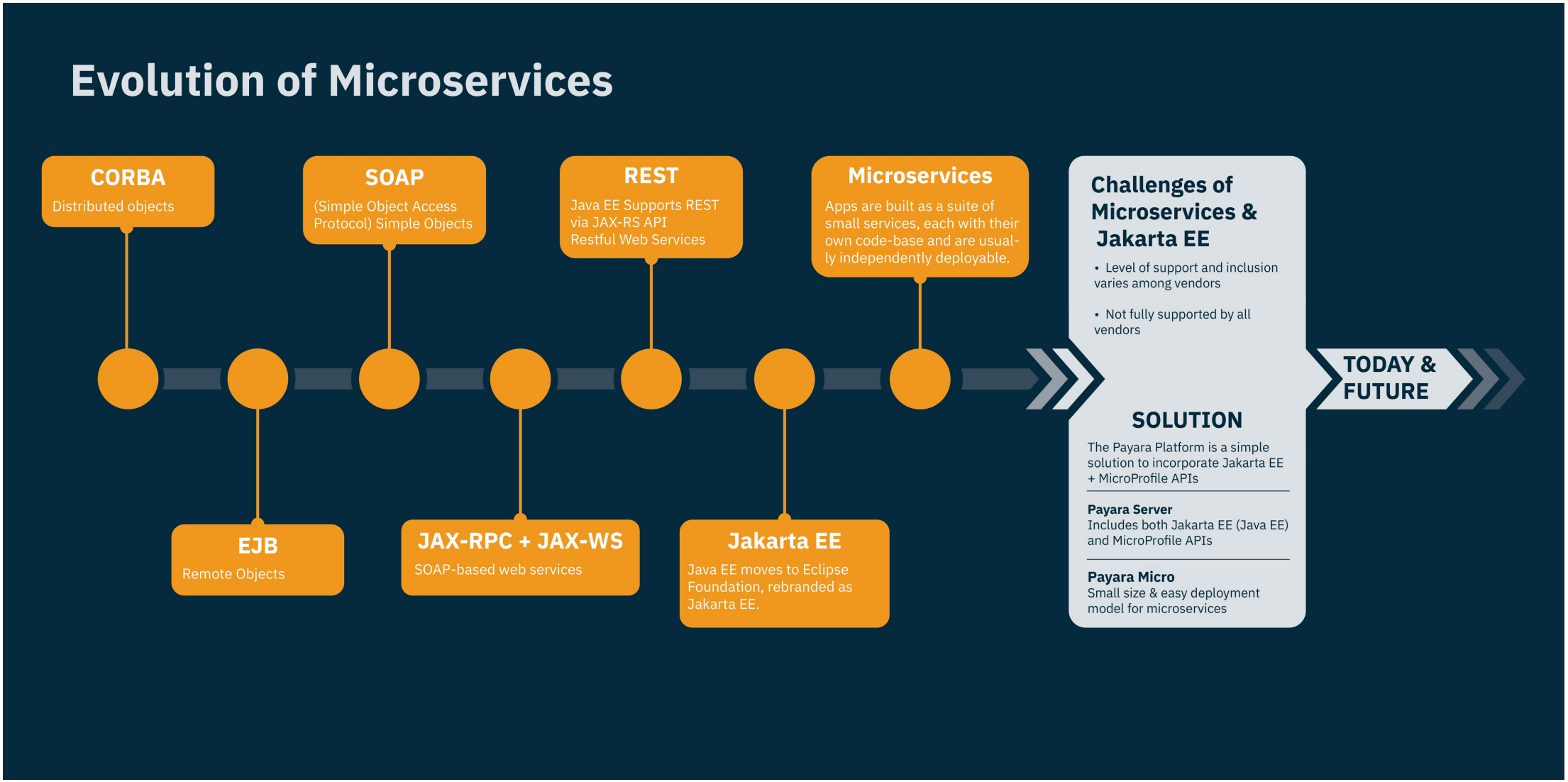




https://cloudblogs.microsoft.com/opensource/2019/10/16/announcing-dapr-open-source-project-build-microservice-applications/

Exercise Using DAPR for building a microservice architecture

- Install DAPR on your local development environment
- Design a basic DAPR-based architecture, such as a stateful service or an event driven system.
- Choose one use case (e.g., building a microservices-based application or implementing distributed data storage).
- Develop a simple DAPR application that demonstrates your chosen use case. Document the steps you followed, the use case, some code snippets, and any challenges you encountered.
- Test your DAPR application to ensure it functions as intended. Evaluate the performance and scalability of your application.
- Analyze the strengths and limitations of DAPR for your chosen use case.



https://blog.payara.fish/evolution-of-microservices

Conclusions

Distributed objects bring the benefits of encapsulation and data abstraction to distributed systems, and associated tools and techniques from oo design

Distributed objects therefore represent a significant step forward from previous approaches based directly on the client-server model.

In applying the distributed object approach, however, some limitations have emerged: it is often too complex in practice to use middleware solutions such as CORBA for sophisticated distributed applications and services, particularly when dealing with advanced properties as, for example, dependability (fault tolerance and security).

Conclusions

- Component technologies separate application logic and distributed systems management.
- The explicit identification of dependencies also helps in terms of supporting third party composition of distributed systems.
- We examined the EJB specification which simplifyied distributed systems development through an approach that emphasizes the use of plain old Java objects with the complexities managed declaratively through the use of Java annotations