CT414

Distributed Systems & Co-Operative Computing

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–02–18

Contents

1 Introduction

1.1 Client-Server Architectures

1.1.1 Two-Tier Architectures

A **two-tier client-server architecture** is a client-server architecture wherein a client talks directly to a server, with no intervening server. It is typically used in small environments ($\lesssim 50$ users).

A common development error is to prototype an application in a small, two-tier environment, and then scale up by simply adding more users to the server: this approach will usually result in an ineffective system, as the server becomes overwhelmed. To properly scale to hundreds or thousands of users, it is usually necessary to move to a three-tier architecture.

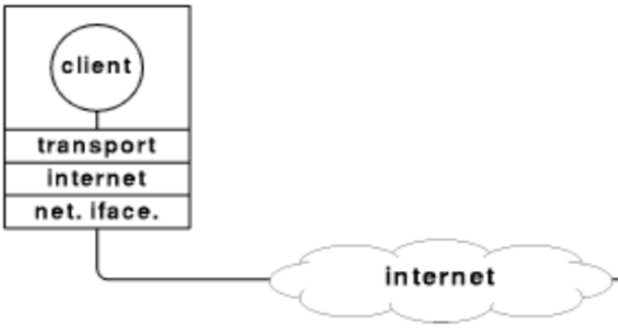

Figure 1: Client & server using TCP/IP protocols to communicate. Information can flow in either or both directions. The client & server can interact with a transport layer protocols.

1.2 Three-Tier Architecture

A **three-tier client-server architecture** introduces a server or **agent** (or **load-balancer**) between the client & the server. The agent has many roles:

- Translation services: such as adapting a legacy application on a mainframe to a client-server environment.
- Metering services: such as acting as a transaction monitor to limit the number of simultaneous requests to a given server.
- Intelligent agent services: as in mapping a request to a number of different servers, collating the results, and returning a single response to the client.

1.3 Network Programming Paradigms

Practically all network programming is based on a client-server model; the only real difference in paradigms is the **level** at which the programmer operates. The sockets API provides direct access to the available transport layer protocols. RPC is a higher-level abstraction that hides some of the lower-level complexities. Other approaches are also possible:

- Sockets are probably the best-known and most widely-used paradigm. However, problems of data incompatibility across platforms can arise.
- RPC libraries aim to solve some of the basic problems with sockets and provide a level of transport independence.
- Neither approach works very well with modern applications (Java RMI and other mdoern technologies, e.g., web services are better).

2 Java RMI

Remote Method Invocation (RMI) is a Java-based mechanism for distributed object computing. RMI enables the distribution of work to other Java objects residing in other processes or on other machines. The objects in one Java Virtual Machine (JVM) are allowed to seamlessly invoke methods on objects in a remote JVM. To call a method of a remote object, we must first get a reference to that object, which can be obtained from the registry name facility or by receiving the reference as an argument or return value of a method call. Clients can call a remote object in a server that itself is a client of another server. Parameters of method calls are passed as serialised objects:

- types are not truncated, and therefore, object-oriented polymorphism is supported;
- parameters are passed by value (deep copy) and therefore object behaviour can be passed.

The Java Object Model is still supported with distributed (remote) objects. A reference to a remote object can be passed to or returned from local & remote objects. Remote object references are passed by reference: therefore, the whole object is not always downloaded. Objects that implement the `Remote` interface are passed as a remote reference, while other objects are passed by value (using object serialisation).

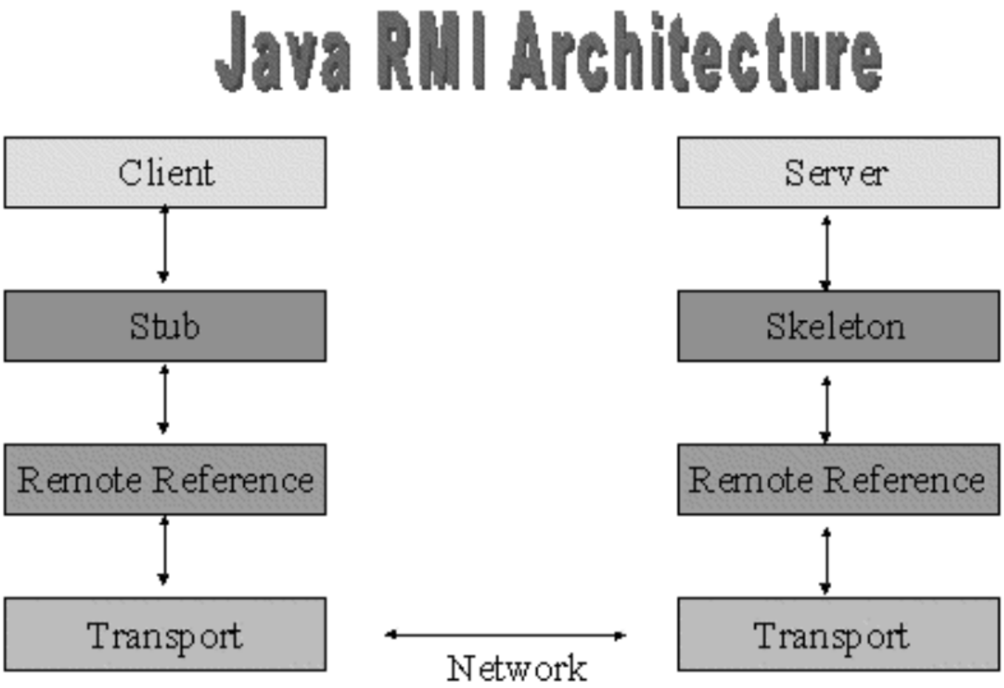

Figure 2: Java RMI Architecture

The client obtains a reference for a remote object by calling `Naming.lookup(//URL/registered_name)` which is a method which returns a reference to another remote object. Methods of the remote object may then be called by the client. This call is actually to the **stub** which represents the remote object. The stub packages the arguments (**marshalling**) into a data stream (to be sent across the network). On the implementation side, the skeleton unmarshals the argument, calls the method, marshals the return value, and sends it back. The stub unmarshals the return value and returns it to the caller. The RMI layer sits on top of the JVM and this allows it to use Java Garbage Collection of remote objects, Java Security (a security manager may be set for the server, now deprecated), and Java class loading.

2.1 Steps to Creating an RMI Application

1. Define the interfaces to your remote objects.
2. Implement the remote object classes.
3. Write the main client & server programs.

4. Create the stub & skeleton classes by running the *rmic* compiler on the remote implementation classes. (No longer needed in later Java versions).
5. Start the *rmiregistry* (if not already started).
6. Start the server application.
7. Start the client (which contains some initial object references).
8. The client application/applet may then call object methods in the remote (server) program.

2.2 Example Java RMI Program

```
// Remote Object has a single method that is passed
// the name of a country and returns the capital city.
import java.rmi.*;

public interface CityServer extends Remote
{
    String getCapital(String Country) throws
        RemoteException;
}
```

Listing 1: Example Java RMI Program

```
import java.rmi.*;
import java.rmi.server.*;

public class CityServerImpl
    extends UnicastRemoteObject
    implements CityServer
{
    // constructor is required in RMI
    CityServerImpl() throws RemoteException
    {
        super(); // call the parent constructor
    }

    // Remote method we are implementing!
    public String getCapital(String country) throws
        RemoteException
    {
        System.out.println("Sending return string now - country requested: " + country);
        if (country.toLowerCase().compareTo("usa") == 0)
            return "Washington";
        else if (country.toLowerCase().compareTo("ireland") == 0)
            return "Dublin";
        else if (country.toLowerCase().compareTo("france") == 0)
            return "Paris";
        return "Don't know that one!";
    }

    // main is required because the server is standalone
    public static void main(String args[])
```

```
    {
        try
        {
            // First reset our Security manager
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager set");

            // Create an instance of the local object
            CityServerImpl cityServer = new CityServerImpl();
            System.out.println("Instance of City Server created");

            // Put the server object into the Registry
            Naming.rebind("Capitals", cityServer);
            System.out.println("Name rebind completed");
            System.out.println("Server ready for requests!");
        } catch(Exception exc)
        {
            System.out.println("Error in main - " + exc.toString());
        }
    }
}
```

Listing 2: Example Server Implementation

```
public class CityClient
{
    public static void main (String args[])
    {
        CityServer cities = (CityServer) Naming.lookup("//localhost/Capitals");
        try {
            String capital = cities.getCapital("USA");
            System.out.println(capital);
        } catch (Exception e) {}
    }
}
```

Listing 3: Example Client Implementation

No distributed system can mask communication failures: method semantics should include failure possibilities. Every RMI remote method must declare the exception `RemoteException` in its **`throw`** clause. This exception is thrown when method invocation or return fails. The Java compiler requires the failures to be handled.

When implementing a remote object, the implementation class usually extends the RMI class `UnicastRemoteObject`: this indicates that the implementation class is used to create a single (non-replicated) remote object that uses RMI's default sockets-based transport for communication. If you choose to extend a remote object from a non-remote class, you need to explicitly export the remote object by calling the method `UnicastRemoteObject.exportObject()`.

The main method of the service first needs to create & install a **security manager**, either the `RMISecurityManager` or one that you have defined yourself. A security manager needs to be running so that it can guarantee that the classes loaded do not perform "sensitive" operations. If no security manager is specified, no class loading for RMI classes is allowed, local or otherwise.

TO make classes available via a web server (or your classpath), copy them into your public HTML directory. Alternatively, you could have compiled your files directly into your public HTML directory:

```
javac -d ~/project_dir/public_html City*.java
rmic -d ~/project_dir/public_html  CityServerImpl
```

The files generated by `rmic` (in this case) are: `CityServerImpl_Stub.class` & `CityServerImpl_Skel.class`.

Polymorphic distributed computing is the ability to recognise (at runtime) the actual implementation type of a particular interface. We will use the example of a remote object that is used to computer arbitrary tasks:

- Client sends task object to compute server.
- Compute server runs task and returns result.
- RMI loads task code dynamically in the server.

This example shows polymorphism on the server, but it also works on the client, for example the server returns a particular interface implementation.

Our example task will be a simple interface that defines an arbitrary task to compute:

```
public interface Task extends Serializable
{
    Object run();
}
```

Listing 4: Simple `Task` interface

We will also define a `Remote` interface:

```
import java.rmi.*;

public interface Compute extends Remote
{
    Object runTask(Task t) throws RemoteException;
}
```

Listing 5: Simple `Task` interface

A task may create a `Remote` object on the server and return a reference to that object; the `Remote` object will be garbage-collected when the returned reference is dropped (assuming that no-one else is given a copy of the reference). A task may create a `Serializable` object and return a copy of that object; the original object will be locally garbage-collected when the `Task` ends. If the `Task` creates an object that is neither a `Remote` nor a `Serializable` object, a marshalling exception will be thrown.

```
import java.rmi.*;
import java.rmi.server.*;

public class ComputeServer extends UnicastRemoteObject implements Compute
{
    public ComputeServer() throws RemoteException {}

    public Object runTask(Task t)
    {
        return t.run();
    }
}
```

Listing 6: Compute server implementation

```
public static void main(String args[])
{
    System.setSecurityManager(new RMISecurityManager());
    try
    {
        ComputeServer cs = new ComputeServer();
        Naming.rebind("Computer", cs);
    }
    catch (Exception e)
    {
        // Exception handling
    }
}
```

Listing 7: Compute server implementation

```
public class Pi implements Task
{
    private int places;

    public Pi (int places)
    {
        this.places = places;
    }

    public Object run()
    {
        // Compute Pi
        return result;
    }
}
```

Listing 8: Task to compute π

```
Compute comp = (Compute) Naming.Lookup("//www.t.nuigalway.ie/Computer");

Pi pi = new Pi(100);
Object piResult = comp.runTask(pi);

// print results
```

Listing 9: The client

In conclusion, RMI is flexible and allows us to pass objects (both `Remote` & `Serializable`) by exact type rather than declared type and download code to introduce extended functionality in both client & server. However, it is Java-only and has been superseded by SOAP & REST as the de-facto standards for communicating with remote services. Nonetheless, RMI is still worth learning to help understand concepts around distributed objects & distributed systems architecture.

3 Enterprise Java Beans

3.1 Distributed System Scenario

Imagine a worldwide financial company with 10,000 online customers that wants to add a new currency converter software component that is heavily used with 1,0000 hits/second. The design will consist of the business logic and the distributed infrastructure. The distributed infrastructure includes security, load-balancing, transaction management, & object-relational mapping; Enterprise Java Beans takes care of this, and provides an API & framework.

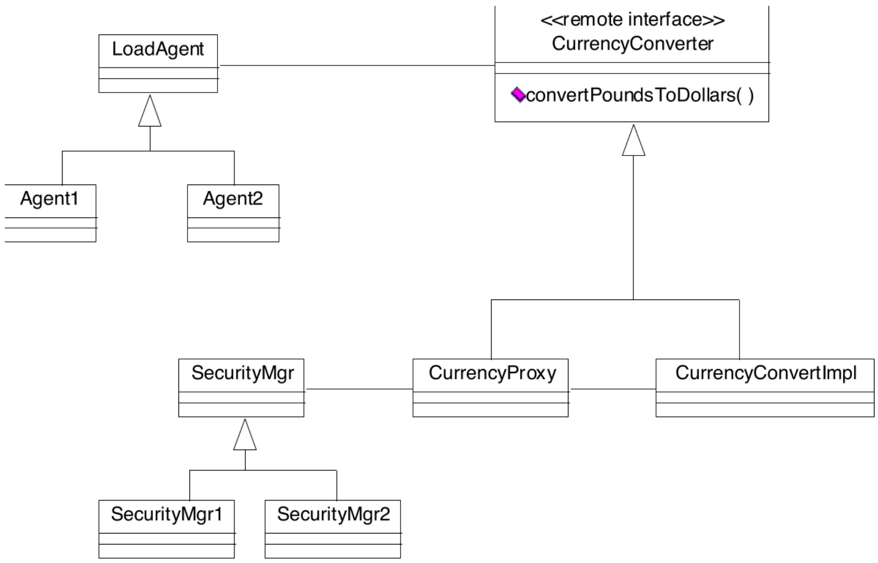

Figure 3: Business logic, distribute the object, add security manager, add load balancing agent.

3.2 EJB

Enterprise Java Beans (EJB) is a server-side component architecture that enables and simplifies the process of building enterprise-class distributed object applications in Java. It allows you to write scalable, reliable, and secure applications without writing your own complex distributed object frameworks. EJB is a *specification*.

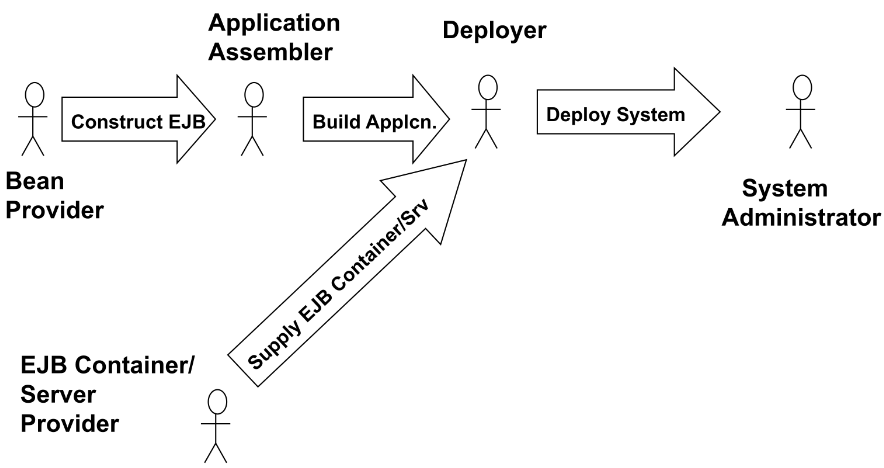

Figure 4: The EJB process

The **EJB Container** is where the EJBs run and is responsible for managing EJBs. The **EJB Server** is a runtime environment for container(s) that manages the low-level system resources.

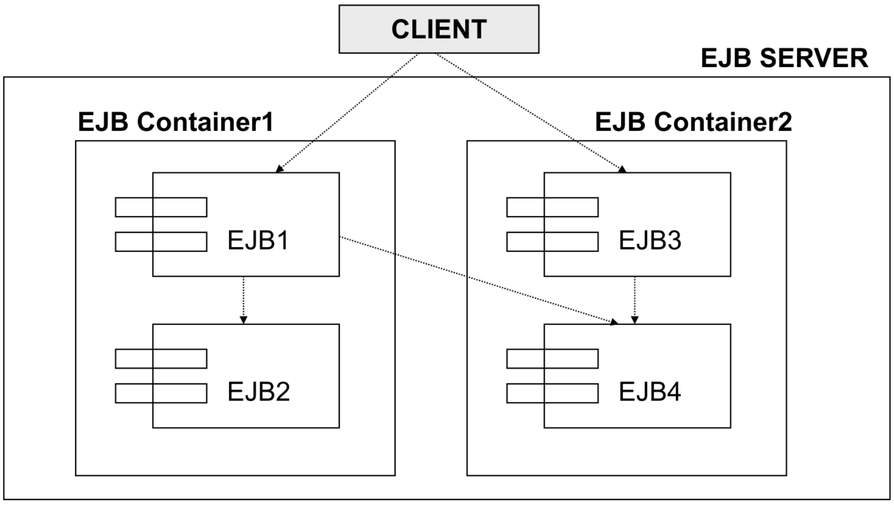

Figure 5: The EJB server & containers

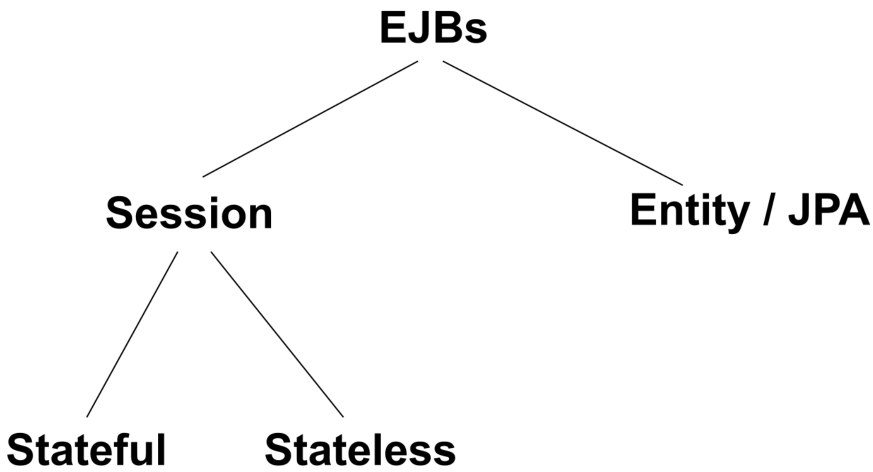

Figure 6: EJ Bean types

Session beans are "business process objects" (e.g., price quoting, order entry, video compression, stock trades, etc.) and live for as long as the client's session. They are usable by 1 client at a time and are *not* shared. The EJB server manages the lifetime of beans. **Stateless session beans** are single request with no state kept, e.g., currency converter, compression utility, or credit card verification.

Entity beans / JPA represent persistent data. They are the object-oriented in-memory view of data in an underlying data store. They are long-lasting and have shared access. Sub-types of entity beans include: bean-managed persisted entity beans and container-manager persistent entity beans. **Bean-managed persistence** must be persisted manually and must look after saving, loading, & finding. They make use of a persistence API such as JDBC or SQL/J. **Container-managed persistence** is automatic persistence wherein the container/server looks after the loading, saving, & finding of component data. You must describe what you want persisted. Deployment tools provide support for defining simple object-relational mappings.

The client never invokes the bean instance, instead it invokes the **EJB object** by an invocation that is intercepted by the container, delegated to the bean instance. The EJB object is a surrogate, network-aware wrapper object that serves as a layer of indirection between the client & the bean; it is essentially the glue between the client & the bean. EJB objects must clone every business method that your bean class exposes, specified in the remote interface. All remote interfaces derive from `javax.ejb.EJBObject`.

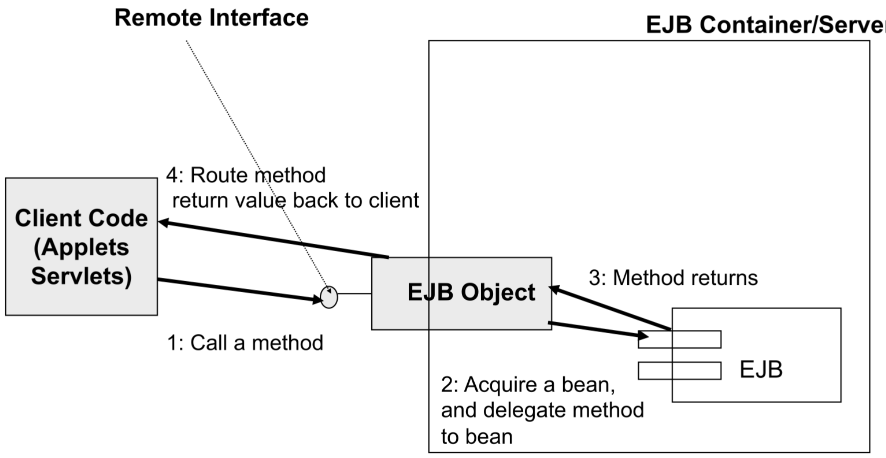

Figure 7: EJB Objects

The **session bean interface** is implemented by all session beans and specifies lifecycle methods that may be implemented inn the bean such as `setSessionContext`, `ejbCreate`, `ejbRemove`, `ejbPassivate`, & `ejbActivate`.

The **Java Naming & Directory Interface** is used to find an object. The resource (e.g., a bean) is associated with a nickname when deploying; clients of this bean can then use this nickname to look up the resource across a deployment. The client code looks up the reference in JNDI and calls business methods on the EJB object.

3.2.1 Entity EJBs

Entity EJBs are object-based representations of information-tier data such as data stored in a relational database. They represent a particular unit of data, e.g., a record in a database. There are two types of entity EJB:

- Bean-managed persistence;
- Container-managed persistence.

4 NodeJS

NodeJS is a JavaScript runtime environment that runs Google Chrome's V8 engine. It is a server-side solution for JavaScript which compiles JavaScript, making it quite fast. It was created in 2009 and designed for high concurrency, without threads or new processes. It has evented I/O for JavaScript, and never blocks, not even for I/O. It's goal is to provide an easy way to build scalable network programs. It provides a JavaScript API to access the network & file system and instead of threads, node uses an event loop with a stack which alleviates the overhead of context switching.

- JavaScript on the server-side ensures that communication between the client and the server will happen in the same language, with native JSON objects on both sides.
- Servers are normally thread-based, but Node is **event-based**; Node serves each request in an evented loop that can handle simultaneous requests.
- Node is a platform for writing JavaScript applications outside web browser, and is therefore not quite the same as the JavaScript we are familiar with in web browser: there is no DOM built-in to Node, nor any other browser capability.
- Node doesn't run in a GUI, but runs in the terminal or as a background process.

Threads	Event-Driven
Lock application / request with listener-workers threads.	Only one thread, which repeatedly fetches an event.
Uses incoming-request model.	Uses queue and then process it.
Multi-threaded server might block the request which might involve multiple events.	Manually saves the state and then goes on to process the next event.
Uses context switching.	No contention and no context switches.
Uses multi-threading environments where the listener & worker threads are used frequently to take an incoming-request lock	Uses asynchronous I/O facilities (callbacks, nor poll/select or `O_NONBLOCK` environments).

Table 1: Threads versus asynchronous event-driven

Ordinarily, a webserver waits for server-side I/O operations to complete while processing a web client request, thus **blocking** the next request to be processed. Servers generally do nothing but I/O, and scripts waiting on I/O requests degrades performance. Node processes each request as an event, it doesn't wait for the I/O operation to complete, making it **non-blocking**; it can therefore handle other requests at the same time. When the I/O operation of the first request is completed, it will callback the server to complete the request. To avoid blocking, Node makes use of the event-driven nature of JavaScript by attaching callbacks to I/O requests. Scripts waiting on I/O waste no space because they get popped off the stack when their non-I/O related code finishes executing.

4.1 MEAN

MEAN is a full stack solution consisting of MongoDB, Express, Angular, & node.

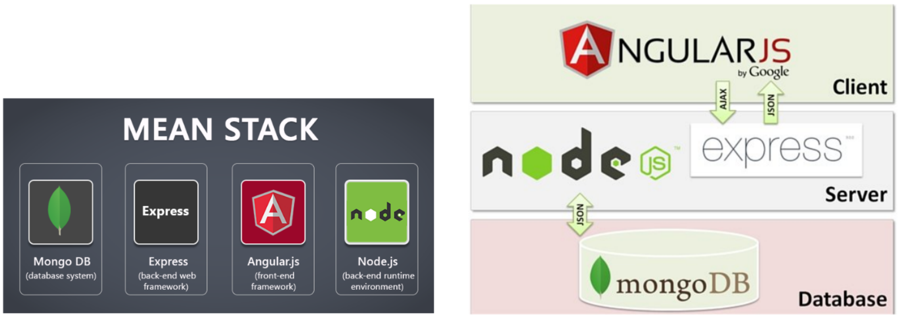

Figure 8: MEAN stack

5 Virtualisation

KVM stuff

QEMU (Quick Emulator) is an open-source hosted hypervisor that performs hardware virtualisation. It emulates CPUs through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems. It uses KVM Hosting mode in Proxmox where QEMU deals with the setting-up and migration of KVM images. It is still involved in the emulation of hardware, but the execution of the guest is done by the KVM as requested by QEMU. It uses the KVM to run virtual machines at near-native speed (requiring hardware virtualisation extensions on x86 machines). When the target architecture is the same as the host architecture, QEMU can make use of KVM particular features, such as acceleration.

LXC (Linux Containers) is an operating-system-level virtualisation method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. The Linux kernel provides the cgroups (control groups) functionality that allows limitation & prioritisation of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines. It provides namespace isolation functionality that allows complete isolation of an application's view of the operating environment, including process tress, networking, user IDs, and mounted file systems. LXC combines the kernel's cgroups and support for isolated namespaces to provide an isolated environment for applications. Docker can also use LXC as one of its execution drivers, enabling image management and providing deployment services.

Ceph is a storage platform that implements object storage on a single distributed computer cluster, and provides interfaces for object-level, block-level, & file-level storage. Ceph aims for completely distributed operation without a single point of failure, scalable to the exabyte level. Ceph's software libraries provide client applications with direct access to the Reliable Autonomic Distributed Object Store (RADOS) object-based storage system. Ceph replicates data and makes it fault-tolerant, using commodity hardware and requiring no specific hardware support. As a result of its design, the system is both self-healing and self-managing, aiming to minimise administration time and other costs. When an application writes data to Ceph using a block device, Ceph automatically striped and replicates the data across the cluster. It works well with the KVM.