***Objectives for 4 weeks starting from 14th April 14, 2014***

1. Closure vs. lambda expressions in terms of accessing variables in cloud
2. Check task parallelism and lambda expressions in C#
3. Check Scala and Clojure in JVM that goes to functional programming
4. Cloud execution services in Java and their provided features
5. Task parallelism oriented environments in different languages
6. Cloud service for mobile phones, their API supports and how to distinguish between tasks in terms of computation vs. communication
7. Starting own projects under ParaTask

# Lambda Expressions:

Lambda expressions do not have *shadowing* effect. Shadowing effect is introduced in nested classes. When a declared type in a particular scope has the same name as another declaration in the enclosing scope, the declaration shadows the declaration of the enclosing scope. Lambda expressions are lexically scoped, that is, they do not introduce a new level of scoping. Lambda expressions enable software developers to treat functionality as method argument. One of the lambda expressions common usages is to simplify the declaration of anonymous classes and functional interfaces.

## Ideal Use Cases for Lambda Expressions

### Sending a Method as Parameter

One of the common cases in which lambda expressions would appear remarkably helpful is when we want to implement an interface with one and only one public method. The helpfulness of lambda expressions becomes even more obvious when we would like to use anonymous (inline) classes for implementing these functional interfaces. As a matter of fact, one of the drawbacks of anonymous classes that would discourage using them is their chunky size, which would consequently cause the method invocation look untidy and hard to understand. Lambda expressions allow us to fit the implementation of a functional interface in only one line, enabling us of sending method specifications as parameters to other methods.

Java 8 enables developers to use the syntax of a lambda expression to implement a functional interface. On the background, by considering the number and types of parameters, the return value of a method and finally the declaration of the method which is receiving the lambda expression as an implementation for a specific interface, the compiler can intelligently understand which functional interface has been implemented by the developer. Consider the following example, where there are two interfaces called “*Predicate”* and *“Consumer”* each of which has only one method. On the other hand there is a method called *“printEligiblePeopel”* which receives three parameters of types “*List<Person>, Predicate<Person> and Consumer<Person>”.* When invoking the method mentioned above we can use lambda expressions at the time of invocation, to provide an implementation for each of the *Predicate* and *Consumer* interfaces as follows.

### Lambda Expressions in GUI Applications

*Public interface Predicate<T> {*

*Public boolean test (T t);*

*}*

*Public interface Consumer<T> {*

*Public void action(T t);*

*}*

*Public class Printer{*

*Public void print(List<Person> list, Predecate<Person> pre, Cosumer<Person> con){*

*For (Person p: list){*

*If(pre.test(p))*

*Con.action(p);*

*}*

*}*

*}*

/\* now we can either have separate implementations for the interfaces used above, or specify their implementation at the time of invoking them using lambda expressions. \*/

//invoking:

*Printer pr = new Printer();*

*Pr.print (list, //list of type Person*

*P -> p.getAge() > 23, //implementation for the only function of ‘Predecate<Person>’*

*P -> System.out.println(“Person’s name: “ + p.getName())*

*); //and last line is implementation for the only function of ‘Consumer<Person>’*

Lambda expressions are also ideal candidates for specifying implementations for functional interfaces that are provided as event handlers for managing the events propagated from the Graphical User Interface.

*Btn.setOnAction(new EventHandler<ActionEvent>( ) {*

*@Override*

*Public void handle(ActionEvent event){System.out.println(“ButtonPressed”);}*

*}); //this is the classic implementation*

*Btn.setOnAction( event -> System.out.println(“ButtonPressed”));*

*//this is the modern approach using lambda expressions*

## Accessing Local Variables of the Enclosing Scope

Like local and anonymous classes, lambda expressions can capture variables, and they have the same access to local variables of the enclosing scope. Starting from Java SE 8, a local class can access local variables and parameters of the enclosing class that are *final* or *effectively final.* A variable or parameter whose value is never changes after it is initialized is effectively final. Once the value of a final local attribute is accessed, that attribute could be called *captured.* In Java SE 8, if we declare the local class in a method, it can access the method’s parameters.

However, unlike local and anonymous classes, lambda expressions do not have any shadowing issues. Lambda expressions are lexically scoped, this means that they do not inherit any names from a supertype, and do not introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are in the enclosing environment.

# Closure:

Closure is a function or a reference to a function together with a referencing environment. In the referencing environment, a table stores a reference to each of the non-local variables (also called *free variables* or *up-values*). A closure, unlike an ordinary function pointer, allows access to non-local variables even when invoked immediately after their *lexical scope.* Closures are closely related to ‘Actors’ in the Actor Model of concurrent computation where the values in the function’s lexical environment are called acquaintance.

An important issue for closures in a *concurrent programming* language is whether the variables in a closure can be updated, and if so, how these updates can be synchronized. Actor Model provides one solution to this. Lambda expressions and closures are normally used when we have methods that are used only once.

# Actor Model

Actor model is a mathematical model of *concurrent computation* that treats ‘actors’ as the universal primitives for concurrent digital computation. For example, in response to a message that it receives, an actor can make local decisions (e.g. create more actors, send more messages and deciding on how to respond to future messages). This model has been used both as a *theoretical understanding of computation,* and as *theoretical basis for several practical implementations of concurrent systems.* This model adopts the philosophy that *everything is an actor* which is similar to *everything is an object* in object oriented design. However, OOP is often executed sequentially, but actor model is inherently concurrent.

An actor is a computational entity that in response to a message it receives, it can concurrently send a finite number of messages, create a finite number of new actors, or designate the behavior to be used for the next messages it receives. There is no assumed sequence to the actions thereof, and they could be carried out in parallel. Decoupling the sender from communications sent was a fundamental advance of the actor model. This enabled *asynchronous communications* and helped controlling the structures as patterns of *passing messages.*

Recipients of messages are identified by address, sometimes called “mailing address”. Thus an actor can only communicate with actors whose addresses it has. The addresses could be obtained from the messages that one actor receives, or the address of the actors that are created by an actor itself. The Actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages and interaction only through direct asynchronous *message passing* with no restriction on message arrival order.

# Functional Programming

In general concepts such as lambda expressions or closures contribute to functional programming. Functional programming is based on lambda calculus (or expression), and is declarative. That is, functions are declared specifying the relation between input and output, and functions always evaluate to the same value for the same arguments (i.e. free of side effects). Lambda expressions can express everything that is computable, and their important feature is that they do not have any states.

A functional programming language that by default excludes destructive modification (to data structures) is called “pure”. All pure functions can be executed in parallel, and a compiler can easily make them fit for multicore processors (i.e. re-arrange order of function execution or inline). Runtime can cache function evaluations. However, I/O is the beast for this concept that needs to be tamed.

Cloud needs to support scalable programs, including any applications that could be scaled through distribution over parallel (multicore) hardware. Applications with high concurrency are good candidates for parallelism. A pure functional programming language like “HASKELL” offers strong functional language, strong types, it is inherently parallel, it is immutable by default, it offers lazy evaluation and code maintainability.

Functions are Data as well, and they return pointer to a function (function name). Function signature declares constraints (types) and computational strategies. Cloud computing and functional programming languages comply in following aspects.

1. Asynchronous Operations – Immutable data, nothing is shared, message passing available to re-synchronize processes. Use of STM (Software Transactional Memory) works better than Locks.
2. Parallel, multi and many cores support – FPLs are inherently parallel using functions, closures and currying. FPLs are declarative, and the compiler has the freedom to re-arrange everything.
3. Elasticity and large scale operation – FPL code is easily testable and maintainable. However, the elasticity is up to the developers.

# Software Transactional Memory (STM)

STM is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. This paradigm is an alternative to lock-based synchronization. A transaction in this context occurs when a piece of code executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time, and intermediate states are not visible to other successful transactions.

Unlike the *locking* technique, STM is very optimistic. That is, a thread completes modifications to shared memory without regard for what other threads might be doing, recording every *read* and *write* in a log. Instead of placing the onus on the writer to make sure it does not affect other operations in progress, it is placed on the reader, who after completing an entire transaction verifies that other threads have not concurrently made changes to memory that is accessed. Changes are finalized through an operation caused *commit,* where the changes of a transaction are validated. If a transaction is not validated it is *aborted* causing all of the prior changes to be rolled back (undone). If a transaction is aborted, it is typically rolled back and re-executed from the beginning until it is successful. This offers the benefit of more concurrency, as no thread has to wait for accessing a resource, and they can all safely read resources simultaneously. However, at the same time this approach introduces the overhead associated with maintaining the log and the time spent committing transactions. Therefore, STM suffers from a performance hit compared to fine-grained lock-based system on small number of processors.

STM also makes multithreading programs more understandable and maintainable. Moreover, STM roles out the following well-known drawbacks introduced by lock-based approaches.

1. Locking requires thinking about overlapping operations and partial operations in distantly separated and seemingly unrelated sections of code.
2. Locking requires programmers to adopt locking policies for preventing dead-locks etc. These policies are fallible and difficult to debug.
3. Locking can lead to priority inversions where a high priority thread is forced to wait for a low priority thread.

In contrast, the concept of memory transaction is much simpler, because each transaction can be viewed in isolation as a single-threaded computation. Dead-lock and live-lock are either preventable, or handled by an external transaction manager. *Priority inversion can still be an issue, but high priority transactions can abort conflicting lower priority transactions that have not already committed.* On the other hand, the need to abort failed transactions also places limitations on the behavior of transactions. That is, transactions cannot perform any operation that cannot be undone including most I/O. Such limitations are typically overcome in practice by creating buffers the queue up irreversible operations and perform them at a later time outside of any transaction.

# Parallel Task (ParaTask)

In ParaTask some the following concepts are used with which it is necessary to be familiar.

## Priority Blocking Queues

Priority blocking queues are a type of priority queues. In a priority queue, elements are ordered according to their natural order, or by a comparator provided at queue construction time. This type attempts to traverse the nodes of a priority heap based on the priority of the nodes.

## Thread Local

Thread local storage is a programming language method that uses *static* or *global* memory local to a thread. This is sometimes required, as normally all threads in a process share the same address space which is sometimes undesirable. In other words, data in a static or global scope is always located at the same memory location, to which threads of the same process refer. With thread local, each thread will make its own copy of the static or global memory.

## Count Down Latch

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. A *CountDownLatch* is initialized with a given *count.* The *await* method blocks until the current count reaches zero due to invocation of the *countdown ()* method, after which all waiting threads are released and any invocation of *await* returns immediately.

A *CountDownLatch* initialized with a count of one, serves as a simple on/off latch, or gate: all threads invoking *await* wait at the gate until it is opened by a thread invoking *countdown ().*  A *CountDownLatch* initialized to *N* can be used to make one thread wait until *N* threads have completed some action, or some action has been completed *N* times. A useful property of a *CountDownLatch* is that it does not require that threads calling *countdown* wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an *await* until all threads could pass.

Count down latch works in latch principle, in which the min thread will wait until the Gate is open. One thread waits for a number of threads specified while creating the *CountDownLatch* in java. Any thread which calls *CountDownLatch.await()* waits until count reaches zero, or it’s interrupted by another thread. All other threads are required to count down by calling *CountDownLatch.countDown()* once they are completed.

One of the disadvantages of *CountDownLatch* is that it is not reusable, and once the *count* reaches to zero, the *CountDownLatch* cannot be used anymore. *Cyclic Barrier* covers this problem and allows users to use it more than once. Count down latch is used whenever we have a thread that needs to wait until a number of other threads have finished processing their tasks.

# New Objectives Added to the Previous Ones

1. Learn about java reflections.
2. Learn about Proxy Pattern.
3. Learn about what jar files are in java projects.
4. Learn about Apache Ant and creating jar files.
5. Learn about Reduction in programming
6. Learn about how to work with SVN for windows and Linux.
7. Learn about making plugins for ‘Eclipse’
8. Learn about Orphaned methods (noWait).
9. Learn about WorkSharing VS. WorkStealing approaches.
10. Stat on the Lyx tutorials.

# Call Reflection

In computer programming, reflection is the ability of a computer program to examine and modify the structure and behavior (specifically values, meta-data, properties and functions) of the program at runtime. Reflection is mostly used in high-level virtual machine programming languages (e.g. SmallTalk), and also statically typed programming languages (e.g. Java, Haskell, C#, Scala). A language supporting reflection provides a number of features available at runtime that would otherwise be very obscure to accomplish in a lower-level language. Some of these features are abilities to:

1. Discover and modify source code construction (such as code blocks, classes, methods, protocols, etc.) as a first-class object at run-time.
2. Convert a string matching the symbolic name of a class or function into a reference to or invocation of that class or method.
3. Evaluate a method as if it were a source code statement at runtime.
4. Create a new interpreter for the language’s byte-code to give a new meaning or purpose for a programming construct.

Reflection can be implemented for languages not having built-in reflection facilities by using a *program transformation* system to define automated source code changes.

**Example**

*//without reflection*

Foo foo = new Foo();

foo.hello();

*//with reflection*

Object foo = Class.forName(“complete.classpath.and.Foo”).newInstance();

*//alternatively: Object foo = Foo.class.newInstance();*

Method m = foo.getClass().getDeclaredMethod(“hello”, new Class<?>[0]);

m.invoke(foo);

**Reflection in Java**

Reflection is commonly used by the programmers to examine or modify the runtime behavior of applications running in the Java Virtual Machine. Reflection is a powerful technique and can enable application to perform operations which could otherwise be impossible.

***Extensibility Features***

An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.

***Class Browsers and Visual Development Environments***

A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.

***Debuggers and Test Tools***

Debuggers need to be able to examine private numbers on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suit.

***Drawbacks of Reflection***

Reflection can cause ***performance overhead****,* as it involves types that are dynamically resolved and therefore certain JVM optimizations cannot be performed. Consequently, reflective operations have slower performance and should be avoided in parts of the code that are called frequently in performance-sensitive applications.

Moreover, reflection requires a runtime permission which may not be present when running under a security manager, and therefore would cause ***security restrictions***. This is an important consideration for code which has to run in a restricted security context, such as in an Applet (applet is any small application that performs one specific task that runs within the scope of a dedicated widget engine or a large program.

# Classes

Every object is either a reference or a primitive type. Reference types all inherit from “java.lang.Object”. Classes, enums, arrays and interfaces are all reference types, and there is a fixed set of primitive types: *Boolean, byte, short, int, long, char, float* and *double.* Reference types also include, string, all of the wrapper classes such as Double, the interface Serializable and the enum SortOrder.

For every type of object, the Java virtual machine instantiates an immutable instance of “java.lang.Class”, which provides methods to examine the runtime properties of the object including its members and type information. The entry point to all reflection operations is “java.lang.Class”, with the exception of “java.lang.reflect.ReflectPermission”. To get to these classes it is necessary to invoke appropriate methods on “Class” depending on whether the code has access to an object, the name of class, a type, or an existing “Class”.

1. If an instance of an object is available, we use the method *Object.getClass().* Arrays are objects as well and we can use this method on them. In this case, the returned *Class* corresponds to an array with component type *byte.* For example “java.util.set” is an interface to an object of type “java.util.HashSet”. The value returned by *getClass()* is the class corresponding to “java.util.HashSet”.
2. If the type is available, but there is not instance of the type, we can obtain the class by appending “.class” to the name of the type.
   1. boolean b;

Class c = b.getClass(); //compile-time error

Class c = boolean.class; //correct

1. If the fully qualified name of a class is available, it is possible to get the corresponding Class using “Class.forName()”. However, this method cannot be used for primitive types.

**NOTE:**

The most convenient way to access the Class of a primitive type is using the “.class” syntax, as it is mentioned above. However, each of the primitive types has a *wrapper* class (as well as void) in “java.lang” which is used for boxing of primitive types to reference types. Each wrapper class contains a field name called “TYPE” which is the equivalent to the “Class”.

Class c = Double.TYPE; ---🡪 identical to “ Class c = double.class”

# Members

Reflection defines an interface called “java.lang.reflect.memebr” which is implemented by “java.lang.reflect.Field”, “java.lang.reflect.Method” and “java.lang.reflect.Constructor”.

## Fields

Fields have types and values. “Java.lang.reflect.Field” provides methods for accessing type information and setting and getting values of a field on a given object.

1. Field f = className.getField(stringName);
2. String typeName = f.getType();
3. String genericTypeName = f.getGenericType();

We can also retrieve and parse field modifiers. There are different types of modifiers for java fields and variables, which could be retrieved using “*Field.getModifiers()”.*

1. Access Modifiers: *protected, private, public*
2. Field specific modifiers governing run-time behavior: *transient, volatile*
3. Modifier restricting to one instance (*static),* and modifier prohibiting value modification (*final)*, and annotations.

Once we have access to an instance of a class, we can use reflection to set the values of its fields.

*Class<?> c = book.getClass();*

*Method[ ] allMethods = c.getDeclaredMethods();*

*For (Method m: allMethods){*

*if (!m.getName().equals(args[1])){continue;}*

*print(“Return Type:” + m.getReturnType());*

*print(“Generic return type:” + m.getGenericReturnType());*

*Class<?>[ ] pType = m.getParameterTypes();*

*Type[ ] gpType = m.getGenericParameterTypes();*

*for (int i = 0; i < pType.Length(); i++){*

*print(“parameter type:” + pType[i]);*

*print(“generic parameter type:” + gpType[i] );*

*}*

*Class<?>[ ] xType = m.getExceptionTypes();*

*Type[ ] gxType = m.getGenericExceptionTypes();*

*for (int i = 0; i < xType.Length(); i++){*

*print(“Exception type:”+ xType[i]);*

*print(“Generic exception type:”+gxType[i]);*

*}*

*}*

## Methods

Methods have return values, parameters and may throw exceptions. The “java.lang.reflect.Method” class provides methods for obtaining the information about the parameters and return value of a certain method, and even invoking methods on a given object.

*Class<?> c = book.getClass();*

*Field chap = c.getDeclaredField(“chapters”); //chapters is the name of a field in class ‘book’.*

*chap.setLong(book, 12); // setting the value of ‘chapters’ to 12*

*Field chars = c.getDeclaredField(“characters”);//characters is a list of string specifying the characters*

*String[ ] names = {“Queen”, “King”};*

*chars.set(book, names); //setting new names for the characters of the book*

*Field t = c.getDeclaredField(“twin”); // twin is a field as a type of ‘Tweedle’ enum*

*t.set(book, Tweedle.DUM); //setting the enum type of ‘twin’ to “DUM”*

We can also obtain the names of the formal parameters of any method or constructor using the “java.lang.reflect.Executable.getParameters” method, as classes *Method* and *Constructor* extend class *Executable,* so they inherit the method.

# Proxy Patterns

In general Proxy is an agent or substitute authorized to act for another person, or a document which authorizes the agent so to act.

**Proxy Patterns**

In computer programming, proxy pattern is a software design pattern, and in its most general form, it is a class functioning as an interface to something else. A proxy could interface to anything (e.g. a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

# Jar Files

In software **JAR** (**J**ava **Ar**chive) is a package file format typically used to aggregate many *java class files* and associated *Meta data* and *resources* (such as text, image, etc.) into one distribute *application software* or *libraries* on the java platform. JAR files are fundamental archive files, built on the ZIP file format.

# Apache Ant

Apache Ant is a software tool for automating the software build process. It is similar to *Make,* but is implemented using the *java* language, requires the java platform, and is best suited to building java projects. The most noticeable difference between ‘ant’ files and ‘make’ files is that ‘Ant’ uses *XML* to describe the build process and its dependencies, whereas ‘Make’ uses *Makefile format.* By default the XML file is named ***build.xml***.

Because ‘Ant’ made it trivial to integrate *junit* tests with the build process, Ant made it easy for willing developers to adopt *test-driven development* and even *extreme-programming.*

**Test Driven Development**

In the test-driven development (TDD) the software development process relies on the repetition of a very short development cycle. First, the developer writes an (initially failing) automated test (that defines a desired improvement or new function), then produces the minimum amount of code to pass the test. Furthermore, the code is refactored and improved the meet the specifications of the project.

**Extreme Programming**

Extreme Programming (XP) is a software development methodology, which is intended to improved software quality and responsiveness to changing customer requirements. As a type of *Agile-Software-Development* it advocates frequent releases in short development cycles.

Ant files are used to automate compiling code, packaging binaries, deploying binaries to the test server, testing changes and copying code from one location to another. Typically Ant build files are called ‘*build.xml’* and they live in the project’s base directory by default.

*<?xml version=”1.0”?>*

*<project name=”Hello World Project” default=”info”>*

*<target name=”info”>*

*<echo>Hello world – Welcome to Apache Ant! </echo>*

*</target>*

*</project>*

Note that there should be no blank lines or white space before the XML declaration. All build files require the project element, and at least one target element. The project element has the following attributes.

1. **name**: the Name of the project (optional).
2. **default**: the default target for the build script. A project may contain any number of targets. This attribute specifies which target needs to be considered as default (Mandatory).
3. **basedir:** The base directory (or) the root folder for the project (Optional).

The target is a collection of tasks that we want to run as a unit. Targets may have dependencies on other targets. For example, a **deploy** target may have dependency on the **package** target. Dependencies are denoted using the **depends** attribute. A target element is consisted of the following attributes.

1. **name:** the name of the target (required).
2. **depends:** comma separated list of all targets, that this target depends on (optional).
3. **description:** a short description about the target (optional).
4. **if:** allows the execution of a target based on the trueness of a conditional attribute (optional).
5. **unless:** adds the target to the dependency list of the specified Extension Point. An extension point is similar to a target, but it does not have any tasks (optional).

As Ant build files are written in XML, they do not support variables; however Ant uses ***property*** elementto allow specifying properties that can change from one build to another, or from one environment to another. By default Ant provides the following pre-defined properties that can be used in the build files.

1. Ant.file – the full location of the build file.
2. Ant.version – the version of the apache ant installer.
3. Basedir – the base directory of the build, as specified in the **basedir** attribute of the **project** element.
4. Ant.java.version – the version of the JDK that is used by Ant.
5. Ant.project.name – the name of the project, as specified in the **name** attribute of the **project** element.
6. Ant.project.default-target – the default target for the current project.
7. Ant.proejct.invoked-trargets – comma separated list of the targets that were invoked in the current project.
8. Ant.core.lib – the full location of the ant jar file.
9. Ant.home – the home directory of Ant installation.
10. Ant.library.dir – the home directory for Ant library files – typically ***ANT\_HOME/lib*** folder.

In addition to the properties mentioned above, Ant also allows users to define additional properties using the ***property*** element.

*<?xml version=”1.0”?>*

*<project name=”Hello World Project” default=”info”>*

*<property name=”siteName” value= ”*[*www.tutorialspoint.com*](http://www.tutorialspoint.com)*” />*

*<target name=”info”>*

*<echo>Apache Ant version is: ${ant.version} – You are at ${siteName} </echo>*

*</target>*

*</project>*

Apache allows storing the properties in a separate property file. This feature becomes very helpful, when dealing with big projects specifically. Storing the properties in a separate file allows reusing the same build file, but with different property settings. Normally the property file is called ***build.properties*** and is placed alongside the ***build.xml*** file. We can create multiple property files based on the deployment environment (e.g. ***build.properties.dev*** and ***build.properties.test***).

*<?xml version=”1.0”?>*

*<project name=”Hello World Project” default=”info”>*

*<property file=”build.properties” />*

*<target name=”info”>*

*<echo>Apache Ant version is: ${ant.version} – You are at ${siteName} </echo>*

*</target>*

*</project>*

**build.properties**

# *The site name*

*siteName =* [*www.tutorialspoint.com*](http://www.tutorialspoint.com)

*buildversion = 3.3.2*

The content of a build property file is similar to the normal *java* property file; it contains one property per line, and each property is represented by a **name** and a **value** pair. The name and the value pair are separated by an equal sign. Properties could be annotated with proper comments using the *hash* character. And also provides a number of pre-defined data types.

1. **File Set –** the file set data types represent a collection of files. This data type is usually used as a filter to include and exclude files that match a particular pattern.

<*fileset dir=”${src}” casesensitiv=”yes”>*

*<include name=”\*\*/\*.java” />*

*<exclude name=”\*\*/\*Stub\*” />*

*</fileset>*

The file-set above, selects all the ‘java’ files in the source folder except those that contain the work ‘Stub’ in them. When the case-sensitive filter is ‘on’, it is applied when selecting the files, such that a file called ***Sampelstub.java*** won’t be selected, but a file called ***SampeStub.java*** will be selected.

1. **Pattern Set –** allows filtering files or folders based on a certain pattern. Pattern sets can be created using *meta-characters*. For example:
   * **? –** Matches one character only.
   * **\* –** Matches zero or many characters.
   * \*\* **–** Matches zero or many directories recursively.
2. **File List –** the file list acts similar to file set, except it explicitly lists the names of the files, and does not support wild cards. The other difference is that file list data type can be applied for files that may or may not exist yet.

<*filelist id=”config.files” dir=”${webapp.src.folder}”>*

*<file name=”applicationConfig.xml”/>*

*<file name=”facesConfige.xml”/>*

*<file name=”web.xml”/>*

*<file name=”portlet.xml”/>*

*</filelist>*

<*patternset id=”java.files.without.stubs”>*

*<include name=”\*\*/\*.java” />*

*<exclude name=”\*\*/\*Stub\*” />*

*</patternset>*

*<fileset dir=”${sirc}” casesensitive = “yes”>*

*<patternset refid=”java.files.without.stubs” />*

*</fileset>*

1. **Filter Set –** using a filter set data type with the copy task, we can replace certain text in all files that match the pattern with a replacement value.
2. **Path –** The path data type is commonly used to represent a *classpath*. Entries in the path are separated using a semicolon or colon. However, these characters are replaced at the runtime by the running system’s path separation character. Most commonly, the classpath is set to the list of jar files, and classes in the project.
   * A class path tells the *java virtual machine* or the *java compiler* where to look for user-defined classes and packages.

<*path id=”build.classpath.jar”>*

*<pathelement path=”${env.J2EE\_HOME}/${j2ee.jar}” />*

*<fileset dir=”lib”>*

*<include name=”\*\*/\*.jar” />*

*</fileset>*

*</path>*

In the snippet above, the ***env.J2EE\_HOME*** refers to the environment variable ***J2EE\_HOME.*** And the ***j2ee.jar*** attribute above points to the name of ***J2EE*** jar file.

## Example

We have a directory with a ‘src’ folder which holds all source codes, a ‘WEB\_INF/lib’ directory which holds all libraries and jar files. Finally we would like to write a build.xml file which compiles the sources files into a folder called ‘WEB\_INF/classes’ which should be created during the build process.

<*?xml version=”1.0” ?>*

*<project name=”fax” basedir=”.” Default=”build”>*

*<property name=”src.dir” value=”src” />*

*<property name=”web.dir” value=”war” />*

*<property name=”build.dir” value=”${web.dir}/WEB\_INF/classes” />*

*<property name=”name” value=”fax” />*

*<path id=”master\_classpath”>*

*<fileset dir=”${web.dir}/WEB\_INF/lib” >*

*<include name=”\*.jar” />*

*</fileset>*

*<pathelement path=”${build.dir}” />*

*</path>*

*<target name=”build” description=”Compile source tree java files” >*

*<mkdir dir=”${build.dir}” />*

*<javac destdir=”${build.dir}” source=”1.5” target=”1.5” >*

*<src path=”${src.dir}” />*

*<classpath refid=”master\_classpath” />*

*</javac>*

*</target>*

*<target name=”clean” description=”Clean output directories” >*

*<delete>*

*<fileset dir=”${build.dir}” >*

*<include name=”\*\*/\*.classes” />*

*</fileset>*

*</delete>*

*</target>*

*</project>*

The next step in building a project is creating the ‘jar’ files after compiling the source code. This step is quite easy using the apache ant. Some of the common attributes used by the ‘jar’ task are as follows.

1. Basedir – the base directory for the output jar file.
2. Compress – advises ant to compress the files as it creates the jar file.
3. Keepcompression – while the *compress* attribute is applicable to individual files, the *keepcompression* attribute does the same job, except that it applies to the entire archive
4. Destfile – the name of the output jar file.
5. Duplicate – advises ant what to do in case of finding duplicate files while creating the ‘.jar’ file. The possible options are ***add, preserve* or *fail.***
6. Excludes – advises ant not to include the comma separated list in the package.
7. Excludesfile – does the same job as above, except that excludes files using a pattern.
8. Includes – inverse of excludes
9. Includesfile – inverse of excludesfile
10. Update – advises ant to overwrite files in the already built JAR file.

<*jar destfile=”${web.dir}/lib/util.jar”*

*Basedir=”${build.dir}/classes”*

*Includes = “faxapp/util/\*\*”*

*Excludes=”\*\*/Test.class”*

*/>*

# Reduction

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. This approach is composed of a ***MAP*** procedure, which involves filtering and sorting (for example, sorting students by first name into queues, one queue for each name). Furthermore, ***REDUCE*** procedure involves performing a summary operation (for example, counting the number of students in each queue). This model is inspired from two commonly used functions in functional programming called ***Map*** and ***Reduce***.

## Map

In many programming languages, map is the name of a higher order function that applies a given function to each element of a list, and finally returning the list of result. In a more general semantic, the term ‘*map’* also applies to a data structure that relates a key to a value, and is capable of holding a list of these keys each of which related to their own specific value where in a more specific case, the value can be a function.

## Reduce

In functional programming ***reduce*** also known as ***fold, accumulate, aggregate*, *compress*** or ***inject*** refers to a higher-order function that analyzes a recursive data structure and recombines the results of recursively processing its constituent parts through use of a given combining operation, and it builds up a return value. For example, calculating the average age of all male members in a collection is a reduction operation.

In java specifically, reduction operations take place by combining the contents of a stream using functions including terminal operations (e.g. *average, sum, min, max, count)*. JDK provides general purpose reduction operations ***reduce*** and ***collect***.

# SVN Tortoise (Subversion)

Subversion is a multi-platform open source version control system. It consists of a repository database (FSFS or BDB) and several command-line tools; however GUI front-ends have been designed for working with SVN as well. TortoiseSVN is a subversion client, implemented as a windows shell extension, a plugin to windows explorer.

**Global Ignore Pattern Settings**

If we don’t want to keep track of temporary and some other types of files, right-click on any folder (or alternatively on desktop) and launch TortoiseSVN/settings. In the ‘general’ section, we can list all ignore file types separated by **spaces**.

**Repository Creation**

On the hard drive (or a server), create the directory for repositories of all of our projects. Right-click on the directory, and choose ‘TortoiseSVN/Create repository here’. Normally we choose the default ‘Naïve File System’, and click the ‘OK’ button. Then the directory will be formed into a repository with some contents such as conf, dav, db, hooks, locks etc.

**Importing our Work Files into Repository**

For this purpose, somewhere on the hard drive we can create a directory with the following sub-directories (branches, tags and trunk). This structure is necessary for more advanced project management. Then we move all the contents of our project into the ‘trunk’ sub-directory. We right-click on the repository, and choose the option ‘import’, and import the directory in which we hold our projects and works. In the URL section, we enter the qualified address of the repository.

**Checking what was imported into the Repository**

For this purpose we right-click on the repository, and start ‘TortoiseSVN/Repo-browser’.

**Creating a Working Directory**

Now that we have the repository with all the files in it, we create an empty directory for keeping our working directory. In order to get a blessed, completely approved and fully loaded subversion directory, we need to check it out from the repository. For this purpose, we have to right-click on the working directory, and choose ‘SVN checkout’. We set the URL to the qualified address of our repository. The blessing tick will appear on the working directory once ‘checkout’ process is finished successfully.

**Sending Changes to Repository**

For sending changes to the repository, we right-click on the selected files, and choose ‘SVN Commit’.

**Adding Files to the Repository**

This process consists of two steps. First, we right-click on the selected files and choose ‘TortoiseSVN/Add’. Second, we right-click on the selected files and choose ‘SVN Commit’.

# Cloud Computing

Cloud computing is the synonym for *distributed computing* over a network, and means the ability to run a program or application on many connected computers at the same time. In cloud computing the seller of the *service* has actual energy-consuming servers which host products and services from a remote location, so end-users can easily have access to the service by connecting to the network without installing anything. The major models of cloud computing are known as *Software as a Service, Platform as a Service* and *Infrastructure as a Service.*

Cloud services may be offered in public, private and hybrid networks. **Google Amazon, IBM, Oracle Cloud, SalesForce, Zoho** and **Microsoft Azure** are some of the well-known cloud vendors. Network-based services are provided on real server hardware and are in fact served up by virtual hardware simulated by software running on one or more real machines. These virtual servers do not physically exist, thus can be moved around and scaled up or down on the fly without affecting the end user.

The focus in Cloud computing is on maximizing the effectiveness of shared resources. The resources are shared by multiple users, and dynamically reallocated per demand. For example, a cloud that serves European users during European business hours (e.g. e-mail) may reallocate the same resources to North American users during the North America’s business hours with the same or different application. This approach should maximize the use of computing power thus reducing the environmental damage as well, since less power, air conditioning, Rackspace, etc. are required for a variety of functions. One of the important features of cloud is that it saves companies from upfront infrastructure costs, and allows them to focus on their business instead of infrastructure.

The main enabling technology for cloud computing is *Virtualization.* Virtualization generalizes the physical infrastructure, which is the most rigid component, and makes it available as a soft component that is easy to use and manage. Users face difficult business problems every day. Cloud computing adopts concepts from *Service-Oriented-Architecture* that can help the user break these problems into *services* that can be integrated to provide a solution. Therefore, cloud computing provides all of its resources as services.

## Cloud Service Models

### Infrastructure as a Service (IaaS)

In the most basic cloud model, the providers offer computer – physical or (more often) virtual machines – and other resources. In order to deploy their applications, the cloud users have to install the supporting operating system images and their application software on the cloud infrastructure. In this model the cloud user patches and maintains the operating system and the application software. The cloud providers typically bill IaaS services on a utility computing basis. Thus, the costs reflect the amount of resources allocated and consumed.

### *Platform as a Service (PaaS)*

In the PaaS, the providers deliver a *computing platform,* which typically includes operating systems, programming language execution environments, databases and web-servers. Application developer can develop their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying software and hardware layers. With some PaaS offers like *Windows Azure,* the underlying computer and storage resources scale automatically to match application demand so that the cloud user does not have to allocate resources manually. The latter has also been proposed by an architecture aiming to facilitate real-time in cloud environments.

### Software as a Service (SaaS)

In the business model using software as a service (SaaS), users are provided access to application software and databases. SaaS is sometimes referred to as “*on-demand-software”* and is usually priced on a pay-per-use basis. In SaaS, cloud providers install and operate application software in the cloud, and users access the software from cloud clients, and they do not manage the cloud infrastructure and platform where the application runs.

Not all computing resources should be loaded onto cloud, as cloud computing is not always cost effective. Cloud computing is beneficial when:

1. Processes, applications and data are largely independent.
2. When points of integration are well defined.
3. Lower level of security works fine
4. The core enterprise architecture is healthy
5. The web is the desired platform
6. The cost is an issue, and the applications are new

Conversely, cloud computing is not recommended when:

1. Processes, applications and data are largely coupled
2. The points of integration are not defined
3. High level of security is required
4. The core internal enterprise architecture needs work
5. The application requires a native interface
6. The cost is an issue, and the application is a legacy

The code in cloud is distributed among several cores/machines and it is asynchronous. That is, we don’t block the code when waiting for WS responses, and the callback comes in any time. Therefore, we always have to take care of ‘current state’, and when callbacks come in we need to decide what to do in dependence of the ‘current state’. ***Functional Programming*** eliminates the ‘state’; therefore it can help with this issue. Thus, in cloud computing the code is stateless, distributed and asynchronous. Thus, functional programming can help with that, but at the same time knowledge of object oriented programming is a must in order to model the world.

At this age sequential execution is over, instead we’re in the age of distribution, parallelization, asynchronous programming and concurrency. The programs are parallel, asynchronous and the programs have to react to external events. The codes and applications might be distributed over several cores, nodes or it might be just server side – client side separation where some code is waiting for response from web service, and some other code waiting for response from the user on the client side. As a developer we have to handle the synchronization. In cloud computing we need to think about the global variable which is in charge of holding the information about the ‘current state’, so that when a response from a web-service arrives, we can decide and take the right action. ***It is this maintenance of the global state which is particularly difficult in asynchronous programming.***

Functional languages can help with this fact (provided in C#, F#, Scala…) as they eliminate the concept of ‘global state’, so the results of methods are guaranteed no matter what the actual state is. Functional languages provide immutable, distributable structures such as collections. Since there is no global state, we can postpone the execution and evaluation of methods till the results are needed.

## Cloud Characteristics

Regardless of their types all cloud computing solutions have several common characteristics.

1. **Managed by the Provider –** knowledge of individual hardware characteristics and capacity measures are no longer important to the developers.
2. **Flexibility of Resource Assignment –** organizations can pay only for the level of use needed. Cloud computing services can alternatively add CPUs or expand available database file storage when needed, and then shrink back when storm of access has passed.
3. **Network Accessibility –** cloud services facilitate access by mobile customers, and remote office locations. Because cloud computing vendors can be located anywhere they can host organizational services from areas outside geopolitical turmoil or environmental threats.
4. **Sustainability –** Allows adjusting power and cooling requirements based on usage requirements.
5. **Manage through Self-Service on Demand –** customers can simply request the cloud provided for a new resource allocation as long as the organization contractual limits on resources allows.

Development of distributed application designs using a standardized application programming interface (API) allows one computer to host an application, while others could hold the data and perform secondary tasks. For example a distributed processing helps with managing bids and items that are spread across many systems. No single system could handle the volume of transactions occurring simultaneously.

Even the location and type of hardware supporting a software application can shift from moment to moment. The cloud is interconnected through standard APIs and XML web service interfaces, which allow developers to rapidly move their application into the cloud without requiring a completely new set of skills. This improves future planning and not worrying about technology’s fast evolution. However, APIs still vary from one cloud provider to another.

Cloud computing also makes extensive use of server virtualization to better utilize cloud hosting servers by allowing multiple systems to run on a more powerful server (*multi-tenancy).* Therefore, it ensures system resources are fully utilized before another server is brought online, further reducing operating costs and data center cooling expenses.

## Using High Performance Computing Models for Distributed Processes Across Multiple Systems

Cloud computing also adopts the concept of High Performance Computing (HPC) by separating individual procedures into multiple simultaneous processes which are sent to individual computers. Furthermore, individual results are combined to provide the complete final result. Some cloud service providers offer high performance computing power on demand for data intensive analytics and modelling, allowing thousands of CPU cores to be available for research.

## Different Cloud Clients

1. Work Stations (thick clients)
2. Thin Clients
3. Servers
4. Mobile Clients – mobile devices and tablets are perfect clients for cloud, as they have sufficient onboard storage for rich user interface applications, but they are limited in CPU power. Thus they rely on remote servers for heavy computational tasks in data processing and analytic applications. Using wireless networking for remote connections, these devices provide excellent clients for modern workforce.

The evolution of traditional data center infrastructure into the cloud starts from server virtualization. That is, concentrating the data center resources across smaller number of physical hosts with a lower percentage of potential resources left idle while consuming power without being used. The evolution moves through privately hosted and hybrid clouds, and finally ends into the fully public cloud infrastructures with all elements virtualized. When deploying cloud services for an enterprise, the installers must identify the expectations for control and management based on the type of cloud and its level categorization (i.e. SaaS, PaaS or IaaS). Different cloud models are as follows.

Virtualized Servers. Virtualized data centers gain a measure of hardware independence allowing the organization to purchase best-cost alternatives. For the same reason the capability to improve disaster recovery and business continuity can be enhanced, as the systems use virtual servers.

Distributed Virtualization. This model is designed by extending the virtualization to include distributed resources using technologies that can transfer operations between automated systems.

Private Clouds**.** Through the implementation of local private cloud resident upon hardware located in local data centers but running cloud infrastructural software. The cloud software provides a standard platform for application development and availability even when the hardware remains heterogeneous in make and model. This is the first true transformation from traditional data center resources to cloud-based alternatives. This model enhances the flexibility of resource allocation, while still relying on local servers. In terms of the business side, mild efficiencies of scale can be achieved, but costs remain both capital and operational, as tech refresh comes only form the organization itself.

Hybrid Clouds. This model was created by bridging between local private clouds and other cloud offering to create hybrid cloud. This model allows organizations to better ensure the availability of their resources during the peak periods. Moreover, this model allows organizations to retain total control over data resources that are critical, sensitive or transformative to their business operations while transferring less-sensitive operations to more efficient public cloud service providers.

## Cloud Service Models

As it was discussed earlier, services provided in cloud could be divided into three main categories of Iaas, Paas and SaaS. Normally the IaaS service is used by system administrators and enterprise planners. This level of consumer is more interested in traditional settings such as networking and storage. The application developers normally use the second PaaS, which can also support hosting infrastructure. End users will consume applications provided by the SaaS. SaaS includes components of both platform and infrastructure services beyond the customers’ visibility.

### SaaS

The number of cloud computing providers decreases rapidly as the level of client control over customization, configuration and management of cloud resources increases. SaaS products are generally pre-built and consumed using the provided functionality without significant customization. However, SaaS is offering remarkable advantages over traditional desktop applications such that the tendency towards using cloud applications is rapidly growing in both providers and the users.

Traditional software requires the users to accept certain expenses specifically for deploying the software and maintaining it up-dated. These expenses would normally include making sure that the requirements for software are met, installing the software on client computer, continuous update and maintenance of software.

### PaaS

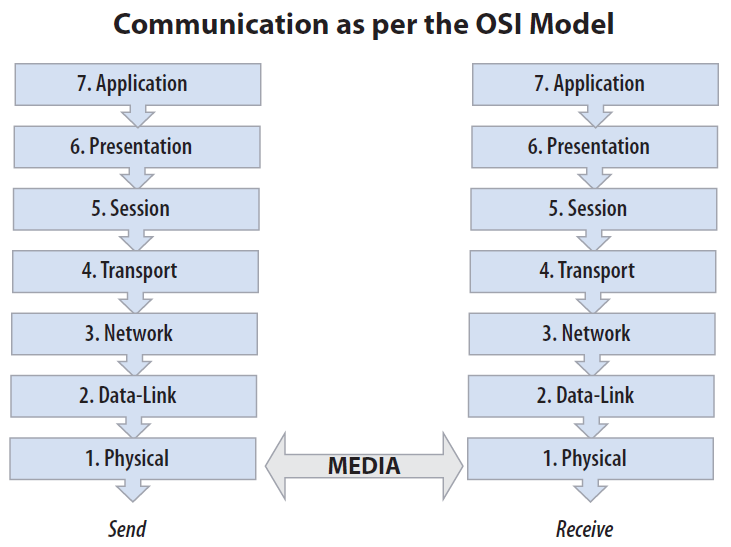
PaaS on the other hand, is normally coupled to a particular vendor’s technologies, languages and other features. As an example, Microsoft Visual Studio is now linked to Azure cloud application service platform to allow its developers use the tools provided in cloud (e.g. high performance computing or servers) in order to develop traditional software applications. Consequently, because the same technology can be used for both traditional and PaaS cloud applications moving the applications from traditional desktops to PaaS cloud is an easy process for enterprises. Hosted on hybrid or public cloud services, PaaS applications can scale to meet even a global consumer base.

However, developers must be careful, as PaaS vendors can decide about the programming languages and other developing toolkits that are supported by their platforms. This is because, they provide their proprietary supporting systems (including operating systems). Thus, it would not be possible for a developer to migrate their application to another platform without additional costs – Vendor Lock In. Most of the leading providers in the PaaS space are trying to future proof their offerings by providing support for popular open-source programming languages such as Python and Java in order to address concerns regarding vendor lock-in.

### IaaS

Infrastructure as a service allows the client almost complete control over applications, languages and fundamental resources supporting their organizational services such as databases, storage and networking. Infrastructure as a service is sometimes referred to as *Hardware as a Service,*

## Accessing the Cloud

It is necessary to consider that any type of cloud service is accessed through networking. Internal private clouds are considered as *organizational intranet,* while public resources are available via the global internet. Therefore, internet plays a fundamental role in cloud computing as it is the medium or the platform through which many cloud computing services are delivered. A key element for this interconnection is the TCP/IP standard in which each device is assigned a unique numerical address through which data could be sent from and received by the target user device. The services that allow TCP/IP interconnectivity to function include functions, services and protocols that facilitate data transport using various mechanisms such as SMTP (for e-mail), FTP (for file transport) and HTTP/HTTPS (for web access). These protocols and their functions are all grouped into the Open System Interconnection (OSI) model, which specifies a seven-layer network communication.

Before a data is transmitted over the network connection, it is broken down into smaller parts called packets. The packets are then taken through each of the layers demonstrated above, with additional data added to the packets in each layer. After being transmitted to the receiving system through the network media, the packets have to go back up through the same levels in order to get reconstructed. This connectivity is used for each communication within the cloud, regardless of type (i.e. public, private or hybrid) or category (i.e. SaaS, PaaS or IaaS).

### Web Access Architecture

Computing Technology Industry Association (CompTIA) refers to the organization of cloud functions against the OSI model of communication as the ***Web Access Architecture***. This architecture applies to cloud service access in both private (local network) and public (via the internet) configurations, regardless of the diverse tools and mechanisms that can be used for connections, even when the client system is a ***thin client system.***

The thin client system relies on the server based applications and services to take the place of locally stored resources in traditional workstations often providing the most basic types of input and output. A thin client system without a local operating system for connectivity to other services may sometimes be distinguished as an ***ultra-thin client*** or ***zero client***. In these systems, the kernel does nothing but initiating the network connection through which a virtual desktop session can be created on a hosting server.

### Cloud Applications Availability

A major advantage of using cloud SaaS applications from the enterprise perspective is that, users can access their applications from any machine, and not only the workstation on which the application has been installed. This eases tech replacement cycles and workforce flexibility, as a worker can easily move between organizational locations without losing access to their cloud-based resources.

At their easiest cloud SaaS applications exist simply as web-accessible components wrapped up within other applications (e.g. Word with Friends game). These types of applications allow a user to interplay with other users within that application regardless of the tools they use for connections (e.g. laptop, mobile or tablet). This connectivity between players across multiple platforms is handled by the cloud services which are also in charge of background calculation and computations.

## Categories of APIs

* **Ordinary Programming –** this API has nothing to do with cloud.
* **Deployment –** APIs to deploy applications to the cloud. This includes traditional packaging mechanisms in addition to cloud specific techniques.
* **Cloud Services –** APIs to invoke storage, queues and databases.
* **Image and Infrastructure Management –** APIs to find images, start images and reboot instances etc.
* **Internal Interfaces –** APIs for the internal interfaces between parts of a cloud infrastructure.

## Developer Roles

* **Client Application Developers –** write cloud-based applications for end users and use *Cloud Services* APIs.
* **Application Developers –** Write traditional applications that use cloud, and use the *Ordinary Programming* and *Cloud Services* APIs.
* **Deployers –** Package, deploy and maintain applications that use cloud, and use *Deployment, Cloud Services* and *Image/Infrastructure* APIs.
* **Administrators –** Work with applications at multiple levels, and use *Deployment, Cloud Service* and *Image/Infrastructure* APIs.
* **Cloud Providers –** Work with internal parts of the cloud, and use *Internal Interfaces* APIs.

## Steps for Starting a Cloud Application

First, it is recommended to start from manipulating and embracing the SaaS models. For example starting from the services provided by different vendors such as Google or SalesForce.com, and find out about their SaaS offerings. It is recommended to begin with Google Apps and SalesForce.com CRM enterprise software. In the next step we can deploy non-critical applications on the cloud, and proceed further with deploying the enterprise applications.

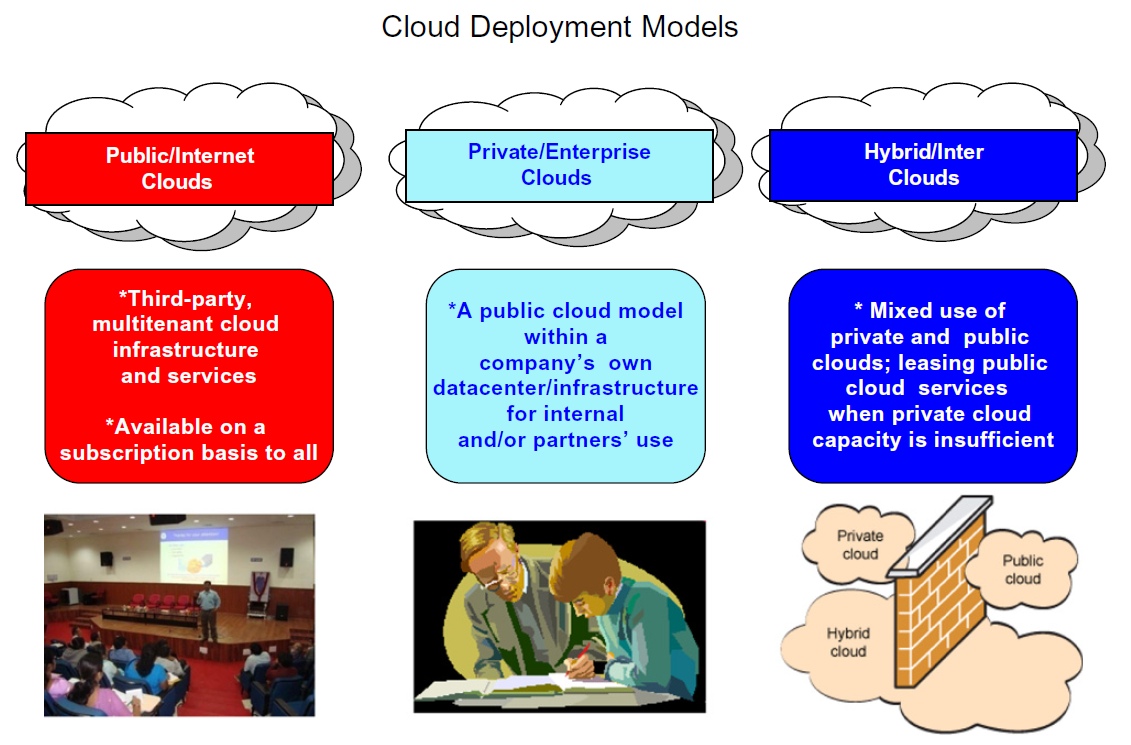


The most general definition of cloud computing brings the idea of everything as service which is normally denoted as XaaS, and it is as follows.

*Cloud computing refers to both the software applications that are provided as a service through the internet, and the hardware and software in the datacenter that provide those services.*

There are five main technologies that have played important roles in realization of cloud computing. They are explained as follows.

1. **Distributed System –** A distributed system is a large collection of independent computers that appears to its user as a single coherent system. Clouds are essentially large distributed computing facilities that make their services available to third parties on demand. This technology that is adopted by cloud computing exhibits properties such as *heterogeneity, openness, scalability, transparency, concurrency, continuous availability and independent failure.*
2. **Virtualization –** virtualization encompasses a collection of solutions allowing the abstraction of some of the fundamental elements for computing such as hardware, runtime environments, storage and networking. Virtualization is essentially a technology that allows creation of different computing environments which are called *virtual* because they simulate the interface that is expected by a guest.
3. **Web –** the web is a primary interface through which cloud computing delivers its services. Web brings interactivity and flexibility providing excellent user experience by offering access to all the functions that are available on normal desktops. These capabilities are obtained by integrating a collection of standards and technologies such as *XML, Asynchronous JavaScript and XML (AJAX), Web services* and others that make the user experience incredibly interactive.
4. **Service Oriented Computing –** Service Orientation is the core reference model for cloud computing systems. This model adopts the concept of services as the main building blocks of applications and system development. A service is an abstraction representing a self-describing component that can do any function (anything from a simple function to a complex business process).
5. **Utility Oriented Computing –** utility computing is a vision of computing that defines a service-provisioning model for computing services in which resources such as storage, computing power, applications and infrastructure are packaged and offered on a pay-per-use basis. The main idea comes from providing computing as a utility like gas, water and electricity.



## Cloud Computing Platforms and Technologies

### Amazon Web Services (AWS)

AWS offers comprehensive cloud IaaS services ranging from virtual compute, storage and networking to complete computing stacks. AWS provides *Elastic Compute Cloud* (EC2) and *Simple Storage Service* (S3) that are well known for compute and storage on demand services. Besides EC2 and S3, a wide range of services can be leveraged to build virtual computing systems including networking support, caching systems, DNS, database (relational or not) support, and others.

### Google AppEngine

Google AppEngine is a scalable runtime environment mostly devoted to executing web applications that would leverage the large computing infrastructure of google to dynamically scale as the demand varies. Google AppEngine provides a secure execution environment and a collection of services that simplify the development of scalable and high-performance web applications. These services include, in memory caching, scalable data storage, job queues and messaging. Developers can build and test applications on their own machines using the AppEngine software development kit (SDK). The languages currently supported by the AppEngine SDK are Python, Java and GO.

### Microsoft Azure

Microsoft Azure is a cloud operating system and a platform for developing applications in the cloud. Similar to Google AppEngine, it provides a scalable runtime environment for Web applications and distributed applications in general. Azure also provides other services such as support for storage (relational data and blobs), networking, caching, content delivery and others.

NOTE – Blob is the term used for data entries that are not of standard primitive data types (e.g. number, date, time, character, string etc.). The word blob is an abbreviation of “Basic Large Object”.

### Hadoop

Apache Hadoop is an open-source framework that is suited for processing large data sets. Hadoop is an implementation of MapReduce, an application programming model developed by Google. Hadoop provides the runtime environment, and developers need only provide the input, and specify the Map and Reduce functions that need to be executed.

### Force.com and SalesForce.com

*Force.com* is a cloud computing platform for developing social enterprise applications. The platform is the basis for *SalesForce.com* a software-as-a-service for customer relationship management. Force.com provides a complete set of ready-to-use blocks that support all enterprise operations. However, it also supports developing individually customized components.

### Manjrasoft Aneka

Manjrasoft Aneka is a cloud application platform for rapid creation of scalable applications, and their deployment on various types of clouds.

## Architectural Styles for Parallel and Distributed Computing

The main goal in parallel and distributed computing is the integrity of data. For this purpose, there are several architecture models offered that are going to be covered in this section.

### Software Architectural Models

**Data Base or Repository –** this model can be divided into two different categories of *databases* and *blackboard* models. The former is consisted of independent components that issue operations on the central data center, and the latter is consisted of a central data center that triggers which process to execute. The black board model is widely used in artificial intelligence applications when the black board model maintains the knowledge about a domain in the form of assertions and results which are entered by the domain experts, and can be divided into two main categories of *Batch Sequential Style* and *Pipe-and-Filter Style*. The former contains a sequence of programs that are known as batches each of which provides the input for the program that is waiting to be run after them, until the output of the last batch provides the main output of the system which is most likely in the form of a file. The latter, differs with *sequential style* from the sense that the individual components of the system which are called *filters* would start processing immediately after another filter provides consumable amount of data in the input stream.

**Virtual Machine Architectural Style –** characterized by an indirection layer between applications and the hosting environment. The major advantage of this design is decoupling applications from the underlying hardware and software environment, and it imposes disadvantages such as slowing down the performance.

**Call and Return Architectures –** systems that are organized into components that are mostly connected to each other by method calls fall into this category. This design can be divided into three main sub-categories of *Top-Down Style, Object Oriented Design and Layered Style.*

Top-Down Style is composed of one large main program that runs from top to bottom and accomplishes its tasks by calling methods. However, this style is hard to maintain and modify when it comes to fairly big projects. In OOP systems are describes in terms of classes and implemented in terms of objects.

The layered style consists of different layers of abstraction each of which is connected with at most two other layers of abstraction. A client interacts with the highest level of abstraction, which in order to carry out its activity interacts and uses the services provided by the lower abstraction levels. Protocols are enforced by interfaces and abstracts. Some of the examples of the layered architecture are the modern operating system kernels, ISO/OSI and TCP/IP.

**Architectural Styles Based on Independent Components –** this architectural style contains systems in terms of independent components that have their own life-cycle, and can be divided into the following sub-categories.

***Communicating Processes*** *–* components are represented by independent processes, and it’s suitable for distributed systems that being distributed over a network of computing nodes and are composed of several concurrent processes. In this design each of the processes provides other processes with services.

***Event Systems*** *–* components are loosely coupled, and each component publishes a collection of events with which other processes can register. During the runtime, and action can activate the event, and consequently the registered callbacks are invoked. Advantage of event-based approach over OOP is that, the invocation pattern between the caller and callee is implicit and not hard coded. Moreover, addition or removal of a handler to an event does not require changes to source code.

### System Architectural Styles

This part is more about the physical organization of the components and processes over a distributed infrastructure. This architectural style can be divided into the following sub-categories.

**Client/Server –** this pattern is very popular in distributed computing. Client and server interact through network using a set of protocols. In this pattern the communication is unidirectional. That is, the client uses request and the server returns a response. There could be multiple clients sending requests to a server that is passively waiting for their requests. This pattern is suitable for many-to-one scenarios, and the most important operations are: request and accept for the client, and listen and response for server side.

The client side per se could be divided into two sub-categories of *thin client* and *fat client.* In the former, the client is only responsible for presenting the data to the user, and is mostly concerned with retrieving and returning data with no further process. However, in the fat client pattern, the client is also responsible for processing and transforming data before returning it to the user, and server is mainly concerned with management of access to data.

There are three major components in the client/server model which are also called tiers, and are as follows, Presentation, Application Logic and Data Storage. In the two tier architecture, the process is divided between the server and the client, which normally imposes scalability and data access limitations on the system. In the three tier/N tier architecture, scalability of the system is considered, as it is possible to distribute the tiers into several computing nodes; however the system could be more complex and harder to understand.

**Peer to Peer Architecture –** in this pattern all components are symmetrically connected, and each component is called a peer and they play the same role by incorporating both client and server capabilities. This architecture pattern is more suitable for highly decentralized architecture, and it provides more scalability. However, managing the implementation of algorithms and the security of data transmission is more complex than the client/server model.

## Models for Interprocess Communications

In distributed systems concurrent processes have to interact over internet; therefore *Internet Protocol Connection* (IPC) is a fundamental aspect. The IPC is mainly used to interchange data and information between the processes, or coordinate their activities. In this regard, sockets are most popular IPC primitives for implementing communication channels between distributed processes. They facilitate interaction pattern by mimicking the client/server pattern in the lowest level based on the request-reply pattern. This powerful abstraction allows system engineers to focus on the logic implementation and the information they would like to exchange rather than fundamental networking details. The most important reference models for architecting the communication among processes are as follows.

### Message-based Communication

Message has played a fundamental role in the evolution of distributed computing technologies, and it applies to any independent amount of information that could be composed from any form of data representation limited in size and time. Some of the popular message-based communications are as follows.

***Message Passing*** *–* message is the main abstraction model, and entities exchange their information explicitly encoded in the form of a message. The structure of a message varies depending on the model (e.g. MPI, OpenMP).

***Remote Procedure Call (RPC)*** *–* involves triggering the execution of code in remote processes based on the client/server model. A remote processor hosts a service on a server allowing client processes to invocate methods of that service and returns results. In this model messages are used as *marshaling* the parameters and return values.

***Distributed Objects*** *–* an implementation of RPC for OOP, where each process registers a set of interfaces that are accessible remotely. On the other side, clients acquire pointers to these interfaces and invoke their methods. The communications between the callers and the remote processors is via message. Because the nature of RPC is stateless, the methods that are remotely executed operate within the context of an instance that is independent from the existence of the requests (e.g. CORBA, javaRMI, .NET Remoting etc.).

***Distributed Agents and Active Objects*** *–* involves the presence of instances which means components have their own control threads, and they explicitly use messages to trigger execution of methods.

***Web Services*** *–* Provides an implementation of RPC over the HTTP. A web service is exposed as a remote object hosted on a web server, and method invocations are transformed into messages called *HTTP Requests.* The requests are packaged using protocols such as *Simple Object Access Protocol (SOAP)* or *Representational State Transfer (REST).*

### Models for Message Based Communication

In general, for every system architecture that is using messages for communication between system components, there must be protocols defined for coordinating and exchanging data. That is, specifying how messages are exchanged, and how many components are allowed to interchange messages. Following are some of the common messaging models used for communication between the nodes in a network; however normally a combination of two or more messaging model is used for managing different communication cases in a network.

**Point to Point Message Model –** Client/Server model is an example of this messaging style. In this messaging style, the communication occurs between individual components (i.e. from one component to another), and there is no central message dispatcher. Therefore, a communication needs to be initiated by a sender, and could be done in two different ways.

1. **Direct Communication –** the message is directly sent to the receptor, and directly processed at the reception time.
2. **Queued Based Communication –** receiver maintains a request queue in which the messages are placed for later processing.

This type of message passing is suitable for one to one, and many to one communication of which Client/Server is a good example.

**Published and Subscribed Model –** This model is based on notification among components. The two major roles in this model are *Publisher* and *Subscriber*, where the publisher will notify the registered subscribers for a specific event, once a set of conditions that indicate the occurrence of that event hold true. This type of message passing is suitable for one to many communication models. In this model the messages could be published based on two different strategies.

1. **Push Strategy –** Publisher notifies the subscribers.
2. **Pull Strategy –** Publisher makes a notifying message ready, and the subscribers check.

**Request – Reply Message Model –** For each message sent by a process, there is a reply. Point to point message models are more likely to be based on a request – reply interaction.

## Technologies for Distributed Computing

**Remote Procedure Call (RPC) –** This paradigm is still the fundamental component for Inter Process Communication (IPC). In this model the called procedure and the calling procedure could be on the same machine, or on different systems in a network. This technology is based on Client/Server model, where a server has registered a set of services that can be remotely invoked, and listens for requests from clients.

The RPC infrastructure is in charge of *marshaling* and *un-marshaling* the messages that are exchanged between the components of a network. Moreover, RPC runtime is in charge of parameter packaging and un-packaging, as well as handling the *request-reply* interactions. RPC also accommodates distributed object programming and web services since sending parameters by reference is not safe, because the clients and server are mostly located on different addresses.

**Distributed Object Framework –** This technology extends the OOP by allowing objects to be spread through a heterogeneous network, and extends RPC to enable remote method invocations on distributed objects. In this technology the common interaction pattern is as follows.

1. Objects are made available through network using interfaces and registered on a server.
2. Clients acquire a reference to the interface representing the shared object using a give address scheme.
3. A client uses the acquired reference to call methods on the shared active object. Parameters and return values are marshaled following the usual message passing mechanism in RPC.

Distributed object framework uses a mechanism called ***Proxy-Skeleton***, where the proxy and the skeleton always make a pair. Skeleton maintained by server side, is in charge of executing the remotely invoked methods. Proxy exists on the client side allowing client to invoke remoter methods. This mechanism is achieved through polymorphism principle (i.e. interfaces and inheritance), as both proxy and active remote objects need to extend the same interface.

Local method calls on proxy pack the requests and send them to the skeleton. The skeleton then, unpacks the requests and executes the requests on the server. Furthermore, the skeleton packs the results of the method calls using local method calls, and sends the results back to the invoking client.

**Service Oriented Computing –** In this technology distributed systems are organized in terms of services. A service is a component that provides a set of coherent and related functionalities. Interaction with a service is done through its interface. An interface associated with a service is kept as small as possible to simplify interacting with that service. Services are defined in terms of schemes or contracts which can be integrated with other services in order to make a more complex context. A service normally advertises a contract in XML format (for better modifiability), in which it specifies the messages it can receive and send.

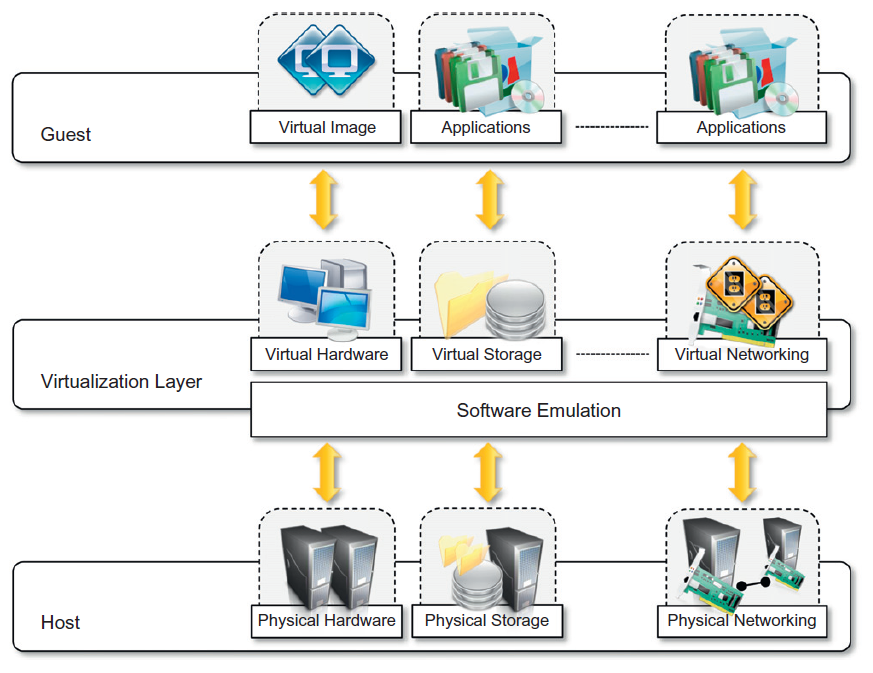
**Service Oriented Architecture –** this architecture organizes a software system in terms of services which leads to a design supporting interconnecting decentralized system, where systems can communicate without having a single point of control. The communications could be stateless due to using services as the fundamental component of the system, which can consequently help with better concurrent programming. The prominent technology that is using SOA is Web Service, where web based services could be invoked using the communication protocols of internet.

### Service Oriented Architecture and Cloud Computing

SOA is suits cloud computing best, as it supports composing and deploying a system as a combination of services, which can consequently ease modification, extension and scalability of the system. Nowadays cloud provides its services via web 2.0 or interacting with AJAX-based clients.

## Virtualization

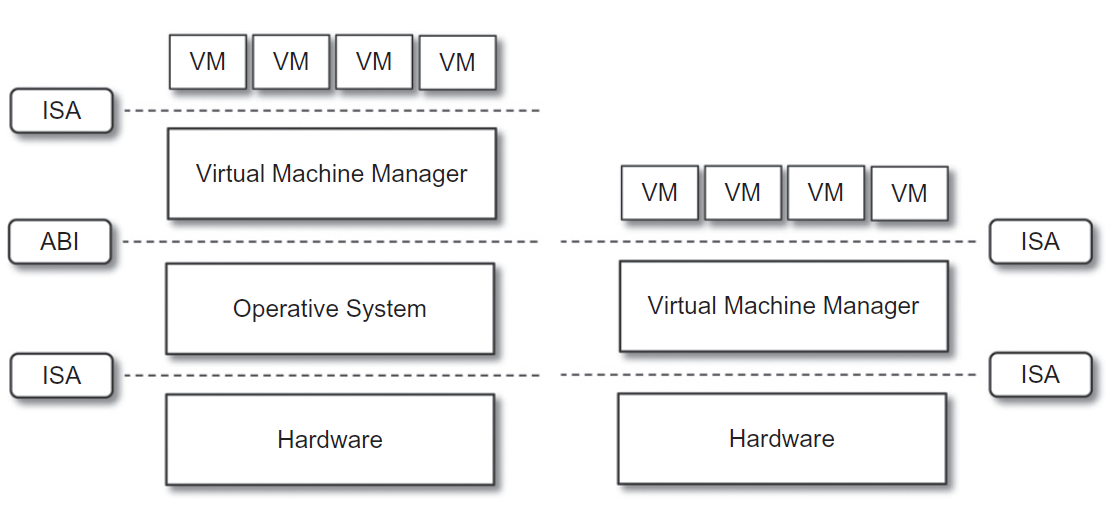
The fundamental aspect for IaaS provision is virtualization. The concept of virtualization could be divided into three sub-categories of *hardware virtualization, storage virtualization* and *network virtualization.* The common point between all three categories is that they are all emulated by software systems. Virtualization specifically helps with security, portability and controllability (managed execution).

 A specific virtual image requires a virtual machine to run on. For example, in programming level binary codes (i.e. jar or assembly files) represent applications and can run on their specified virtual machines (i.e. JVM and .NET in order) without requiring further compilations. Different types of virtualization can be listed as follows.

* *Hardware Virtualization*
* *Operating System Virtualization*
* *Programming Language Level Virtualization*
* *Application Level Virtualization*
* *Application Server Virtualization*
* *Desktop Virtualization*

Hardware virtualization comprises some components that could be listed as follows.

### Hypervisors

There are two types of hypervisors. In the first type, virtual machines run directly on underlying hardware, and communicate with the virtual machine manager via the virtual machine’s Industry Standard Architecture (ISA). Furthermore, the virtual machine emulates the underlying hardware by communicating with the underlying hardware via its ISA. The second type of hypervisors consists of virtual machines that require running on another OS. The virtual machines communicate with the virtual machine manager via the VMM’s ISA; furthermore the VMM communicates the underlying OS via that operating system’s application binary interface (ABI). Virtual machine managers are organized in three main modules as follows.

**Dispatcher –** The entry point to monitor, and reroutes the instructions issued by virtual machines to one of the other two modules.

**Allocation –** Allocates required system resources to a specific VM on demand.

**Interpreter –** Whenever a VM instance executes an instruction, a trap is triggered and the corresponding interpreter routine runs.

A VMM requires the following qualifications in order to efficiently support virtualization.

1. **Equivalence –** A guest application running on a VM should exhibit exactly the same behavior as it runs on the physical host.
2. **Resource Control –** The VMM should have complete control over the virtualized resources.
3. **Efficiency –** A considerable amount of instructions should be executed without the VMM’s intervention.

Desktop virtualization simulates a system/OS that is natively installed on a host; however this host is remotely accessible via network, and makes the desktop ubiquitous. Moreover, software components such as *windows remote service, VNC, X-Server* provide strong infrastructures for supporting multiple virtual desktops on one single server machine.

### Virtualization and Cloud Computing

Virtualization is an important for cloud computing as it allows customization, security, isolation and manageability. Cloud computing per se adopts hardware and programming language virtualization, where the former is mainly used for IaaS level and the latter is mostly used for PaaS level. Virtualization allows cloud software and hardware components to work together in a heterogeneous environment and demonstrate a reasonably high performance.

In general the advantages and disadvantages of virtualization can be listed as follows.

#### Advantages:

* Manageability
* Security
* Portability
* Efficient Use of Resources

#### Disadvantages:

* Performance Degradation
* Unknown Security Holes and New Threats
* Risk of Inefficiency and Degraded User Experience, for example in hardware virtualization level limitations in accessing drivers or their corresponding devices could degrade the user experience.

## Cloud Application Programming

Aneka has a scalable middleware that can be deployed on heterogeneous computing resources. One of the main advantages of Aneka is that, it has a large diversity of APIs (i.e. programming interfaces) that are suitable for a wide range of programming models (Tasks, Threads and MapReduce), and can be used for developing distributed applications or integrating new features into an existing cloud.

### Aneka

Aneka is a software platform for developing cloud computing applications. Aneka is a PaaS solution, a cloud middleware that can be deployed on a heterogeneous set of resources including a network of computers, a multicore server or a data center. The physical and the virtual resources are managed by ***Aneka Container*** which is installed on each node and constitutes the basic building blocks of a middleware. Furthermore, the interconnection of middleware constitutes the ***Aneka Cloud***. Each container provides three different classes of services.

1. **Fabric Service –** This service deals with infrastructure management.
2. **Foundation Service –** This service deals with supporting services for Aneka cloud.
3. **Execution Service –** This service deals with application management and executions.

Aneka implements a service oriented architecture. Each Aneka container allows adding and modifying services. Aneka is a *multitenant* system, in which multiple applications that could potentially belong to different groups run.

### Anatomy of Aneka

The building blocks of the Aneka cloud are the Aneka containers installed in each cloud node. Aneka container is a lightweight software layer, designed to host services and interact with the underlying operating system and hardware. Almost every operation performed in the Aneka cloud is managed by the services provided in Aneka container. The services provided by Aneka container (mentioned above) are installed on a platform abstraction layer which plays the role of an interface to the underlying operating system and hardware. The following paragraphs explain these aspects in more details.

#### Platform Abstract Layer (PAL)

The container’s interface is based on .NET. The platform abstraction layer addresses the heterogeneity of runtime environment, their interfaces to the underlying hardware and OS and file organization system by providing a uniform interface for accessing the underlying infrastructure. PAL has a thin layer of software that automatically detects the hosting environment and configures the Aneka container at the runtime.

#### Fabric Services

These services provide access to the resource provisioning subsystem. Resource provisioning services are in charge of dynamically providing new nodes by relying on virtualization technology. Services that are classified under this category are as follows.

***Profiling and Monitoring*** – This service is done through three major phases of *Heartbeat, Monitoring* and *reporting* services.

1. ***Heartbeat –*** Additional data regarding the information about installed software and dynamic performance is added to *live* messages and published to membership services.
2. ***Monitoring –*** This service forwards all the monitored data that has been collected on one node to the reporting service.
3. ***Reporting –*** This service manages monitored data, and makes them available to other services.

***Resource Management*** – Services under this category provide means for resource membership, resource reservation and resource provisioning.

***Index Service (Membership Catalogue)*** – Services under this category keep track of basic node information for all nodes whether connected or disconnected. A membership catalogue constantly gets updated with information regarding the status of the nodes in a cloud, and is available as a distributed database to ease access for all nodes. Membership catalogue also collects the information regarding the dynamic performance of a node. Dynamic resource provisioning allows the Aneka cloud to scale up and down on demand, ensuring acceptable quality of service (QoS).

***Reservation Service***

***Resource Provisioning*** – Aneka flexibly supports changing the logic of resource provision, having several back-ends and instantiating virtual instances. The implementation is based on ***Resource Pools***, where the resource pool abstracts the interaction with an ***IaaS*** provider via a common interface, so that all pools interact uniformly. A resource pool mostly serves requests coming from the ***Reservation Service*** in order to ensure QoS.

#### Foundation Services

Services under this category provide uniform logical and executional support for the distributed systems that are installed on the infrastructure managed by ***Fabric Services***. Therefore, it allows programs to solely concentrate on the logic of implementation. For better application management, the following services are offered by *Foundation Service.* Foundation service is mainly used by execution services and management consoles.

***Storage Management for Applications*** – Data transfer is normally achieved in the form of files, thus supporting file/data transfer management and reliable data storage is a prerequisite for supporting distributed systems.

***Resource Reservation*** – supports the execution of distributed systems, and is consisted of two different services.

1. **Resource Reservation –** keeps track of all reserved time slots and provides unified view of the system.
2. **Allocation Service –** Installed on each node and manages the execution services and information databases regarding the allocation of slots.

The data storage used for this purpose (resource allocation) is also classified in two different types as follows.

1. **Centralized Database –** More suitable for computationally intensive operations where the number of computing operations is dominant with respect to the data required. Therefore, a centralized data center can be used to mitigate the data retrieval latency.
2. **Distributed Database –** More suitable for data intensive operations, where one data center may not have enough infrastructure (capacity) to support the requirements of the operations.

The centralized storage is managed by ***Aneka’s Storage Service,*** for file transferring and sharing Aneka uses protocols (e.g. File Transfer Protocol). In order to allow variations in protocols, Aneka uses a ***File Channel Controller*** and a ***File Channel Handler.*** The former is used to manage storing and sharing data on the server side; whereas the latter is utilized by client applications to upload, download and browse files. Moreover, Aneka allows concatenating two or more channels with different file transfer protocols.

File systems such as Google File system are based on the distributed file system (i.e. MapReduce), but Aneka is based on the file systems provided by Windows OS as follows.

1. **Basic Reservation –** The basic file reservation system as for most of the applications.
2. **Libra Reservation –** Pricing nodes differently based on their software and hardware infrastructure.
3. **Relay Reservation –** Flexibly manage different logics of different protocols for inter-cloud systems.

#### ***Application Services***

This category of service manages the execution of applications. Types and number of services provided by this layer depends on the specific features of the programming model used for developing a distributed system on top of Aneka; however for most of the models the following two major tasks are commonly supported.

***Scheduling Services*** – These services plan the execution of distributed applications on top of Aneka, and constitute the integration point with several other foundation and fabric services. The main features provided by these services could be listed as follows.

* Job to node mapping
* Rescheduling failed jobs
* Job status monitoring
* Application status monitoring

Each programming model exploits its own defined scheduling service, which consequently gives a great flexibility in customizing scheduling and resource allocation policies, but on the other hand designs need to be accurate and robust, especially in the case of using shared resources. Following are some of the common examples that need to be taken care of.

* Multiple jobs sent to the same node at the same time.
* Jobs without reservations are sent to reserved nodes.
* Jobs sent to nodes where the required services are not installed.

***Execution Services*** – This category of service controls the execution of single jobs that compose applications. Each programming model has its own specifications; however there are common steps among most of the models.

1. Unpacking the jobs received from a scheduler
2. Retrieval of input files required for job execution
3. Sandboxed execution of jobs
4. Submission of output files at the end of the execution of a job
5. Execution failure management
6. Performance monitoring
7. Packing jobs and sending them back to the scheduler

Currently, there are several programming models which need to be supported by application services at the runtime that are listed as follows.

1. **Task Model –** Supports the independent ***bag of tasks*** where an application is designed in the form of a number of independent tasks.
2. **Thread Model –** Uses the abstraction of thread to wrap a method that is being executed remotely.
3. **MapReduce Model –** An implementation of Map/Reduce as proposed by google on top of Aneka.
4. **Parameter Swap Model –** A specialization of a task model for applications that can be described by a template task whose instances are created by generating different combinations of parameters.

### Building Aneka Clouds

Aneka is a tool for developing distributed systems on cloud, thus it needs to be deployed on an infrastructure.

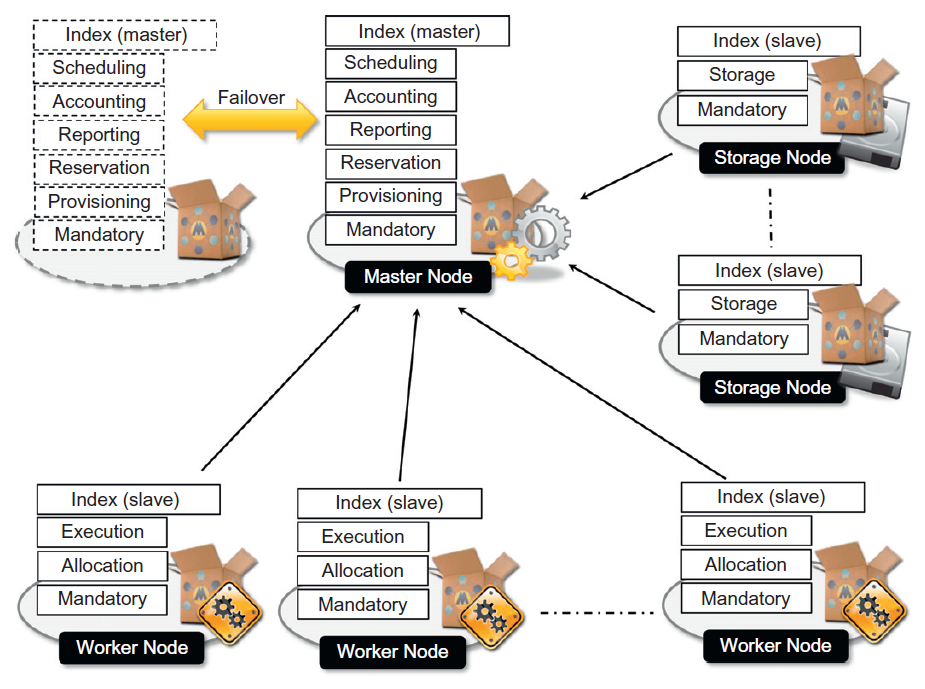
#### Infrastructure Organization

Infrastructure organization provides an overview of the infrastructure of a specific cloud as a reference model for all types of deployments that the mentioned cloud supports. An administrative console is in charge of configuring the required management settings. Management console can manage multiple repositories, and select the best one that suits a specific deployment.

 The infrastructure is deployed by installing Aneka node manager on a collection of nodes (AKA ***Aneka Daemon***). The Aneka daemon is used to remotely deploy and control container instances. Subsequently, the collection of container instances results in Aneka cloud. Virtual and physical nodes can be uniformly managed via an internet connection and a remote administrative access. However, the story is different for dynamic provisioning, as ready to use prepackaged images have the infrastructure installed already and only to need to be configured for joining an Aneka cloud.

#### Logical Infrastructure

The logic depends on the configuration that is set on each of the containers in a specific cloud. In most cases there is one ***Master Worker*** configuration, with separate storage nodes. Master node’s distinctive characteristic is the ***Index Service*** (AKA ***Membership Catalogue***) that is configured in master node. All other services except for the ones that are mandatory could be located in other nodes.

 Master node provides connection to an RDBMS that keeps track of the status of services. Moreover, all scheduling services are kept in the master node. Master configuration can be replicated in several nodes to provide a highly available infrastructure based on the failover mechanism. On the other hand worker nodes contain the mandatory and specific execution services that are required for execution of applications. It is possible to divide the worker nodes into partitions each of which offers a specific set of execution services in order to balance the load between different programming models.

#### Private Cloud Mode

This mode is mostly consisted of local resources and infrastructure management software that provides access to those local nodes which might be virtualized. This type of deployment is suitable for scenarios where the workload could be estimated and the data should be kept encapsulated in a private environment.

#### Public Cloud Mode

This mode features the installation of Aneka master and worker nodes over a completely virtualized infrastructure.

# PaaS Computing

PaaS acts a bit like middleware, where a set of services aimed to help developers with developing and testing their applications without worrying about infrastructure, provisioning resources, storage and backup. PaaS is one level beyond IaaS where the platform, storage and virtual machines are already provided. However, there are differences between PaaS and middleware which are convincing middleware users to gradually move to PaaS cloud.

Middleware is a system that offers useful functionalities such as transactions, security or clustering that are available on remote servers. Middleware allows developers to focus on designing more complex solutions by providing them with ready-to-use functionalities. However, middleware still requires being configured and deployed on the server side statically, and this mainly involves an IT team on the background.

PaaS is a superset of middleware, and besides offering all middleware characteristics, it also provides the operational aspect (i.e. dynamically deploy and configure the application). In other words PaaS provides on demand hardware and operating system and application platforms (e.g. Tomcat, JBoss etc.). Moreover, regardless of the programming language we use, provided that it is supported by the PaaS, there is no need for setting up an execution environment. Because, the PaaS provider offers its own built-in execution engines (runtimes). This characteristic improves portability, scalability and ubiquity of an application.

Java platform could be considered as a suitable option for PaaS, because Java’s virtual machine is a good candidate for an application server. Some of the biggest java middleware companies are IBM, Oracle and VMWare among which VMWare has been slowly moving towards cloud. The following paragraphs list the contemporary popular java PaaS providers.

1. **CloudBees** is a public java based PaaS provider and is hosted in Amazon’s cloud and covers the entire life cycle of an application, from creating the code, testing and continuous deployment (AKA continuous integration) using Jenkins projects that run on the background of a cloud project. As a matter of fact, this continuous integration is CloudBees most remarkable feature.
2. **Amazon Elastic Beanstalk** is the Amazon version of Java PaaS vendor which is based on Amazon EC2, and it offers managed Tomcat instances, complete with load balancer and on demand resource provision. Amazon Elastic Beanstalk integrates with the rest of Amazon in order to provide access to managed relational database management system (RDBMS), big data store, message queues, e-mails and other services.
3. **Google App Engine** is the oldest and by far the most mature Java PaaS vendor. PaaS applications developed under Google App Engine are quite simple and free of unnecessary complications. Google App Engine applications are based on Google web toolkit remote procedure call (GWT RPC) techniques.
4. **Red Hat Open Shift** is the PaaS offering from Red Hat, and it provides web application platforms mainly based on JBoss which is Red Hat’s version of a Java based application server.
5. **Cloud Foundry** is powered by VMWare software which is the basis for most PaaS offerings by virtualizing the data centers.
6. **Heroku for Java** Is the latest offering from PaaS power House Heroku.

## PaaS Fundamentals

Regardless of the PaaS provider, there are some fundamental components commonly used by different PaaS application platforms.

1. **Jenkins Projects –** Jenkins is an open source continuous integration tool in Java. A server based system running in a servlet container. It supports software configuration management (SCM) tools (e.g. subversion, git etc.), and can execute Apache Ant and Apache Maven based projects as well as arbitrary shell scripts and windows batch commands. Builds can be started by various means including being triggered by commit in a version control (SCM) system, scheduling via a ***cron-like*** mechanism or building when other builds have completed.

**Continuous integration** is the process of merging a developer’s working copies with a shared main line (AKA ***trunk***) several times a day. The longer a branch of code remains checked out (i.e. not integrated), the greater the chance of multiple integration conflicts and failures will be.

**Cron mechanism** is a time-based job scheduler in Unix-like computer operating systems. Cron could be used for executing periodic jobs on frequent basis on specific software development environments.

**Apache Maven** is a build automation tool used primarily for java projects. Apache Maven addresses two aspects of building a software system. First, it explains how a software system is built, and secondly specifies what the dependencies are. Apache Maven dynamically downloads java libraries and maven plugins from one or more repositories, and stores them in a local cache. This local cache could also be updated by local projects.

1. **Apache Tomcat –** Apache Tomcat also known as ***Tomcat*** or ***Jakarta Tomcat*** is an open source web server and servlet container. Tomcat implements the java servlet, java server pages, the specifications from ***Sun Microsystems*** and provides a pure ***Java HTTP Web Server*** environment for java code to run in.In the simplest configuration, Tomcat runs in a single operating system process, where the process runs a Java virtual machine (JVM). Every single HTTP request from a client browser to Tomcat is processed in the Tomcat process in a separate thread.
2. **Java Servlet –** Servlet is a Java programming language class used to extend the capabilities of a server. Servlet is commonly used to extend the applications hosted by ***web servers***. Servlets are mostly used for:

* Process or store data that was submitted from an HTML form.
* Provide dynamic content such as the result of a database query.
* Manage state information that does not exist in the stateless HTTP protocol. Servlets can maintain state in session variables using ***HTTP cookies*** or ***URL rewriting***. To deploy and run a servlet, a web container is used which is responsible for the life-cycle of a servlet.

1. **Java Server Page (JSP) –** JSP files are Java Server Pages mainly written in HTML or XML. JSP files are mainly used for dynamically representing data. However, at the background, a JSP file is ultimately converted to a servlet file by a web server.

## Java Servlet

A servlet is a java programming language class used to extend the capabilities of a server. Servlets can be used for responding to any request, but they are specifically used for extending the applications that are hosted by ***web servers***. Java servlets could be thought of as ***Java applet***, which is a small application written in Java and delivered to the user in the form of a bite-code. Thus, in this specific case a user triggers a java applet from a web page, and then it is executed within a JVM in a process separated from the web server process.

Usually a server receives a request for a client in form of an HTTP request; it processes the request and returns the result in form of a static HTML file. Typically an HTTP request contains three key elements.

* HTTP method (the action to be performed) – mainly ***get*** or ***post***.
* The URL of the page to access
* Form parameters (e.g. entries received from the input fields)

On the other hand, an HTTP response has the following key characteristics.

* A status code indicating whether the request was processed successfully (200 = OK, 401 = Page not found, 404 = Failure).
* Content type which is used to inform the client browser about the content of the response (i.e. image, video, text, html etc.).
* The actual content of the response.