Software Architecture

Institute of Computer Science & Information Technology,
Faculty of Management & Computer Sciences,
The University of Agriculture, Peshawar, Pakistan.

Software Architecture

- The architecture of a software system is a metaphor, analogous to the architecture of a building.

- Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems

Software Architecture (1)

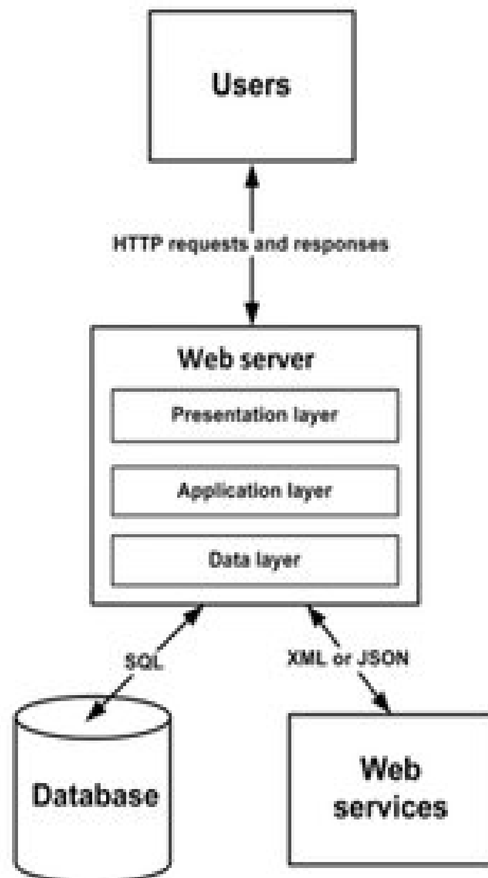
- The architecture of a software system consists of it structures, the decomposition into components, and their interfaces and relationships.

- It describes both the **static** and **dynamic** aspects of the software system, so that it can be considered a building design and flow chart for a software product.

Software Architecture - Views

1. The **conceptual view**, which identifies entities and their relationship;
2. The **runtime view,** the components at system runtime, e.g., servers, or communication connections;
3. The **process view**, which maps processes at system runtime, while looking at aspects like synchronization and concurrency;
4. The **implementation view**, which describes the systems software artifacts, e.g., subsystems, components, or source code.

Software Architecture – Web Server as an Example

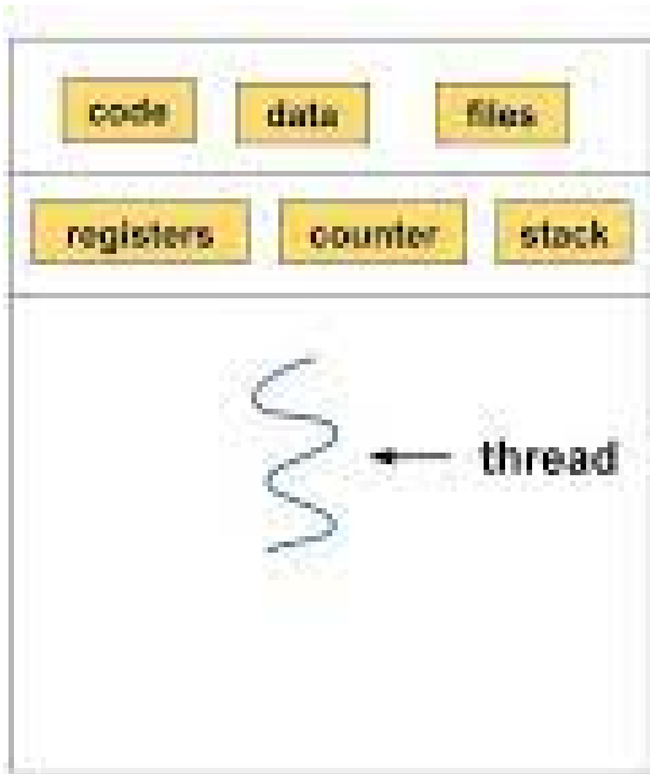


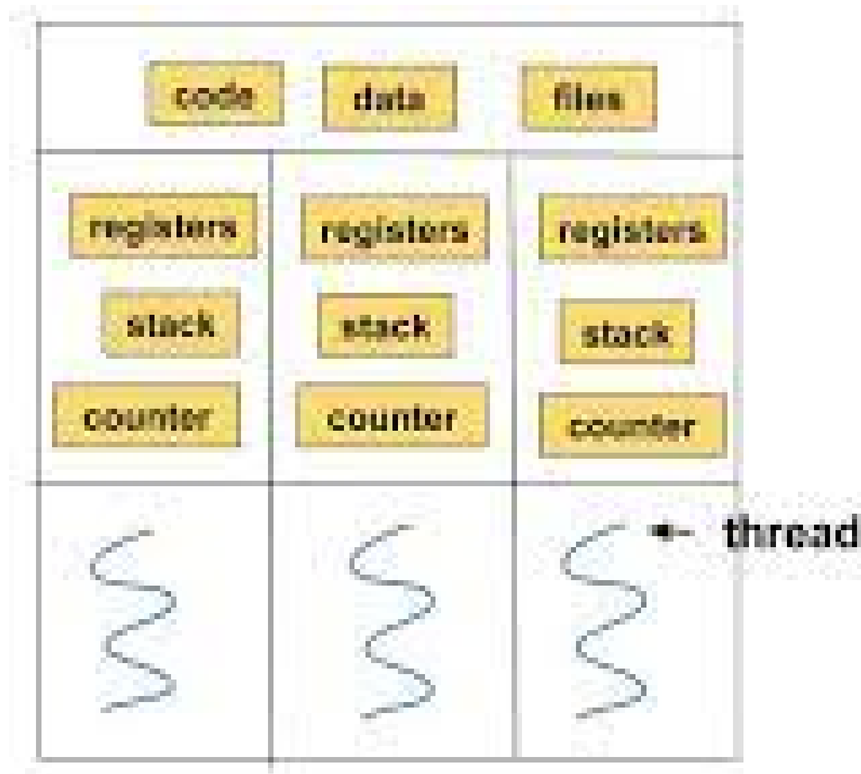
1. **Users:** Make requests to the web server and receive responses using JavaServer Pages (JSPs).
2. **Web server:** Hosts the application's various *layers* which conform with MVC:
   - **Presentation layer:** Users interact with the application via HTTP requests and responses rendered in a browser.
   - **Application layer:** Manages the flow of the application, implements business logic and liaises with the data layer to process requests from users and their responses. Open-source, third-party products reside here.
   - **Data layer:** Handles domain data and provides persistence and retrieval services for the database.
3. **Database:** Where data is persisted and retrieved.
4. **Web services:** Interaction with other applications.

Process and Thread

- Process: A program is in execution is called process.

- Thread: is the segment of a process, mean a process can have multiple threads and these multiple threads are contained within a process.


code
data
files
registers
counter
stack
thread
Single-threaded process


code
data
files
registers
registers
registers
stack
stack
stack
counter
counter
counter
thread
Multi-threaded process

Example

- For example in a word processor, a thread may check spelling and grammar while another thread processes user input (keystrokes), while yet another third thread loads images from the hard drive, and a fourth does periodic automatic backup of the file being edited.

Process	Thread
A process is the instance of a program executed by one or many threads.	A thread is a basic unit of CPU utilisation, which consists of its own thread ID, a program counter, a register and a stack.
A process may contain multiple threads depending on the operating system.	A thread is the smallest unit of execution within a process.
In a multiprocessing environment, multiple processes do not share resources such as memory with each other.	Multiple threads of a given process running concurrently can share resources such as memory with each other.
Processes take more time for context switching.	Threads take less time for context switching.
Processes take more time for creation and termination.	Threads take lesser time for creation and termination.
Processes consume more resources.	Threads consume fewer resources.
No other process can execute until the first process gets unblocked.	Another thread in the same task can run while one thread is blocked and waiting.
Process communication is complex.	Thread communication is much easier and efficient.

Multithreading

- In computer architecture, multithreading is the ability of single CPU to provide multiple threads of execution concurrently, supported by the operating system.
- Multithreading aims to increase utilization of a single core by using thread-level parallelism

Multithreading (1)

- Multithreading allow for multiple request to be satisfied simultaneously, without having to service requests sequentially.

Process and Message Passing

- Numerous programming languages(message passing paradigm) and libraries have been developed for explicit parallel programming.
- The message passing programming paradigm is one of the oldest and most widely used approaches for programming parallel computers.

Process and Message Passing (1)

- There are two key attributes that characterize the message-passing programming paradigm.

- The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelization.

Process and Message Passing (2)

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own address space.

- Instances of such a view come naturally from clustered workstations and non-shared address space multi-computers.

Structure of Message-Passing Programs

- Message-passing programs are often written using the:

1. **Asynchronous** paradigm: all the concurrent tasks execute asynchronously (no coordination) and

- make it possibly to implement any parallel algorithm.
- However, such programs can be harder to understand, and can have non deterministic behavior due to race conditions.

Structure of Message-Passing Programs (2)

2. Loosely synchronous programs are a good compromise.

- In such programs, tasks or subsets of tasks synchronized to perform interactions.
- However, between these interactions, tasks execute completely asynchronously.

Building Block: Send & Receive Operations

- The **interactions** are accomplished by sending and receiving messages, the basic operations in the message-passing programming paradigm are send and receive.

- The prototype of these operations are defined as follows

- send(void*sendbuf, int nelems, int dest)
- receive(void*recvbuf, int nelems, int source)
- The sendbuf points to a buffer that stores the data to be sent.
- The recvbuf points to buffer that stores the data to be received.
- The dest is the identifier of the process that receives the data.
- The source is the identifier of the process that send the data

Building Block: Send & Receive Opertions (1)

P0

```
a = 100;
send(&a, 1,1);
a=0;
```

P1

```
Receive(&a,1,0)
Printf("%d\n",a);
```

- The important thing to note is that process P0 changes the value of a to 0 immediately following the send.
- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.

Building Block: Send & Receive Operations (1)

- Most message passing platforms have additional hardware support for sending and receiving messages.

1. They may support DMA(Direct Memory Access) and
2. Asynchronous message transfer using network interface hardware.

- Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention.

- Similarly, DMA allows copying of data from one memory location to another(e.g., communication buffers) without CPU support (once they have been programmed)

Building Block: Send & Receive Operations (2)

- As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process P1 receive the value 0 in a instead of 100.

Blocking Message Passing Operations

- A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so.

- It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanism by which this can be achieved.

Blocking Message Passing Operations (1)

- There are two mechanisms by which this can be achieved:

1. Blocking non-buffered send/receive
2. Blocking buffered send/receive

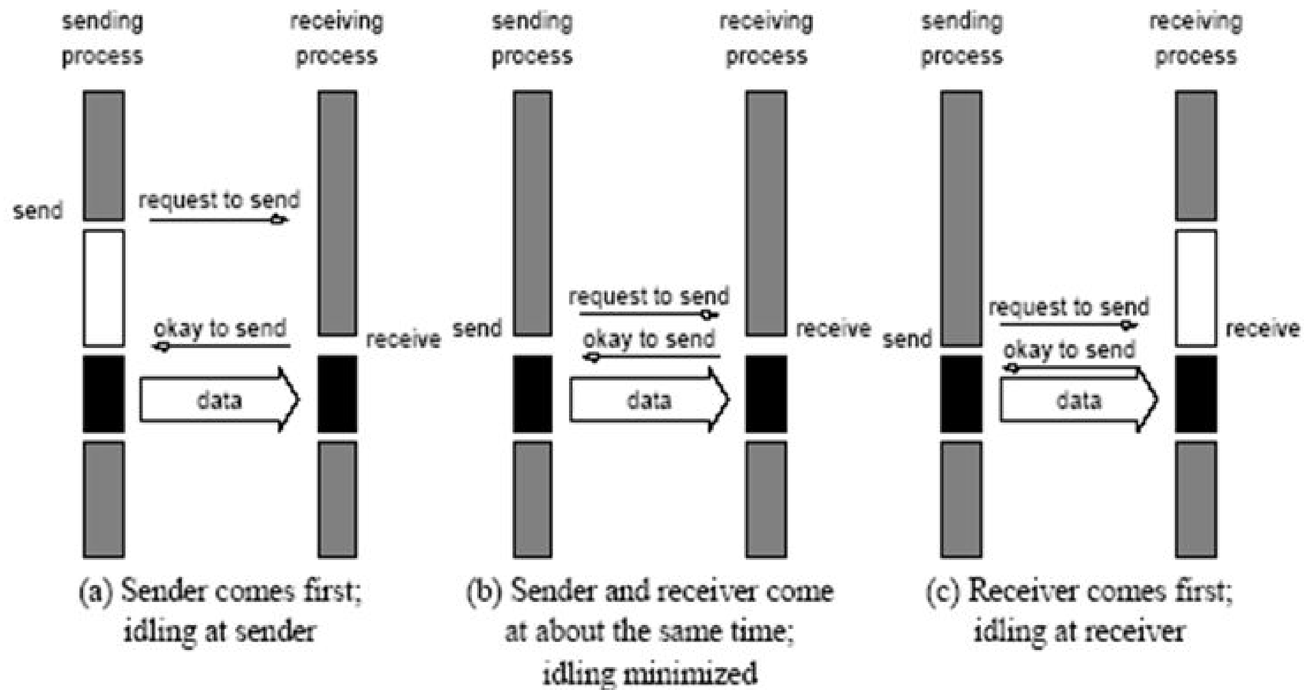
Blocking non-buffered Send/Receive

- In the first case, the send operation does not return until the matching receive has been encountered at the receiving process.

- When this happens, the message is sent and the send operation returns upon completion of the communication operation.

- Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process.
- Since there are no buffers used at either sending or receiving ends, this is also referred to as a non-buffered blocking operations.

Non-Buffered Blocking Message Passing Operations



Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Blocking non-buffered Send/Receive (1)

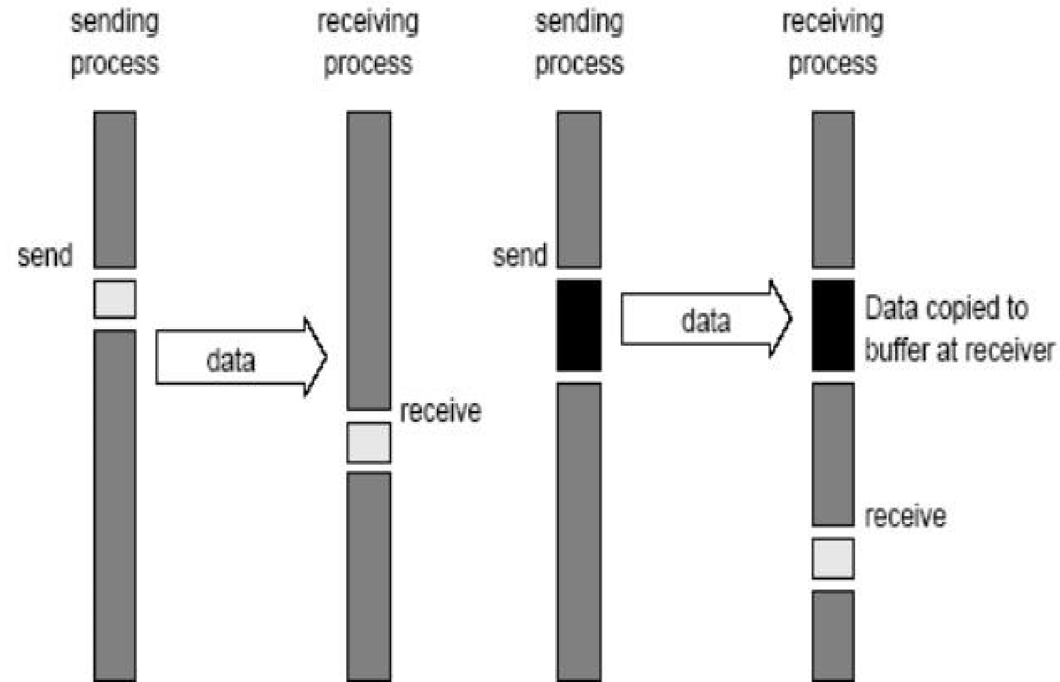
- In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process.
- It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time.
- However, in an asynchronous environment, this may be impossible to predict . This idling overhead is one of the major drawbacks of this protocol.

Blocking Buffered Send/Receive

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communication messages.

- On encountering a send operations, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed.

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Blocking Buffered Send/Receive (1)

- The sender process can now continue with the program knowing that any changes to the data will not impact program semantics.

- Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics.

- Instead, the data is copied into a buffer at the receiver as well.