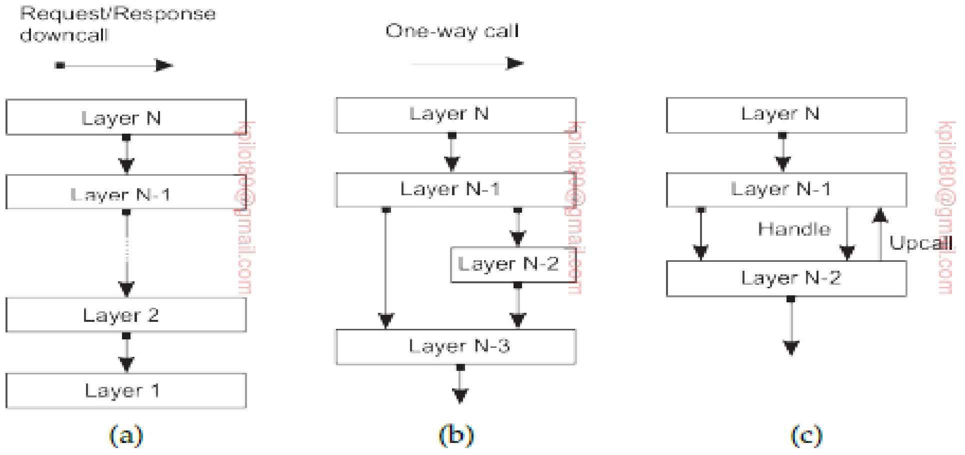
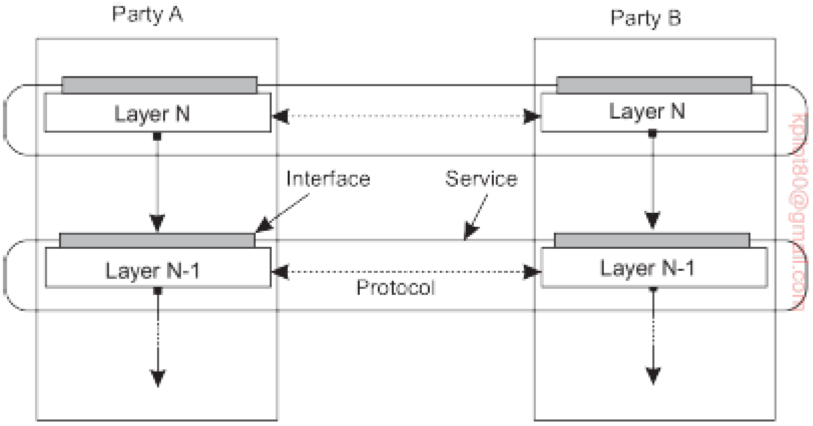
Layered Architecture

- A - pure layered organization
    - Only down calls to the next lower layer are made.
    - In TCP IP, there are 5 layers. (application, transport, network, link, physical) and you can only talk to the next one, not n-2.
    - Layer n-1 will provide an interface to n, and if you want talk to me, you would have to do it through this interface
- B – mixed layered organisation.
    - E.g. take layer n-1. There’s an app called A. this will invoke an OS library that’s available in layer n-3. AS WELL as n-3, A will call layer n-2,wihich holds a maths library . maths library itself relies on OS library in layer n-3! So n-2 has to call n-3 as well.
- C – layered organisation with upcalls
    - Have a lower layer do an upcall to its next higher layer
    - E.g. OS signals the occurrence of an event
        - Its to do with **handle** – Its possible to subscribe to events, and when they become available, it gets an automatic notification.
        - This case, n-1 is interested in event in n-2, so n-2 notifies n-1 that the event (handle) happened, using an upcall.



Layered communication protocols:



- In TCP/IP protocol stack, each layer provides **services** and functions.
- Each layer offers an interface and in order for n to invoke n-1, it exhibits an interface that could be used by layer n. but it has no idea how the functionalities in n-1 is implemented. What's important is the **interface**!!! And that it hides the implementation.
- **Protocol** is a set of rules that parties will follow in order to exchange info. (communication between parties)
- Important to understand the difference between a service offered by a layer.

Object-based and service-oriented architectures

Object based style

- Objects corresponds to components, connected and communicate through a **procedure call** mechanism. If two objects reside on the same system **→ method call,** over a network **→ remote procedure call.**
- Object encapsulate the data **(state),** as they exhibit the interface, but never shows how its implemented.

Client-side stub (**proxy**)

1. Client invokes a method
2. Server gives a copy of **interface** to client → called **proxy**. Proxy is loaded into clients address space.
   - Proxy marshal the method invocation client made
   - and unmarshal reply messages to return the result of the method invocation to the client.
3. The marshalled invocation is passed across **network**.

Server-side stub (**skeleton**)

1. Incoming invocation requests are first sent to a **server stub (skeleton)**
    - Which un-marshals the invocation client sent, and actually make method invocation that client wants at the server through an interface.
    - Also creates a reply and marshals them and forwards reply messages to the client-side **proxy** - **Proxy and the skeleton** are referred to as **stubs**!!!

Resource-based architectures:

- View a distributed system as a **set of resources** where machines, individually manged by components . Resources may be added, deleted, modified, etc by (remote) apps
- Characteristics of **RESTful** architecture:
    - Resources need an identifier→ is usually accessed through **URI (uniform resource identifier**)
    - All the services offer same interface. e.g. put, get, delete, post
    - Messages are fully self-described. E.g. when sending HTML, say that it is HTML. Send its media type!!
    - After executing a service, component forgets about the caller
        - Once the sever gets the request, that server takes the rq, process it, sends back the resource and **forgets it** → is memoryless execution
        - Is prominent in web services and REST
- Operations – put, get, delete, post → **CRUD operation!!**
    - Create (PUT), Read (GET), Update (POST), Delete (DELETE)

E.g. Amazon's Simple storage service (Amazon S3) – RESTful in practice

- **Objects** (=files) are placed into **buckets** (=directories)
- By placing a file in a bucket, file is automatically uploaded to the Amazon cloud
- Object Name contained in Bucket Name –access through: http://BucketName.s3.amazonaws.com/ObjectName
- URI - Operations are carried out by sending HTTP requests –
    - PUT - request through HTTP
    - GET – to see if the object is contained in the Bucket Name
    - S3 – access a file in s3 web service
    - Specific object in that bucket called Object Name
- Simple as long as you know the URI

Event Based Architecture:

- As processes join and leave, its important that dependencies between processes are as loose as possible → hence, architecture that has strong separation between **processing** and **coordination**
  - So more of autonomously operating processes
- Here we emphasize the **coordination**!! → it encompasses the communication and cooperation between processes and machines
- Two design:
  - **Mailbox** coordination- 2 processors, to work and exchange info – they use shared mailbox , and they communicate through this shared mailbox
    - Write and fetch to the mailbox
    - No real communication between the two
  - **Event-based** coordination – coordination between processors will happen once the event occurs
    - Processer 1 **publish a notification** describing the occurrence of event 1, and if you are interested, you **subscribe**, and will be notified!! And will have access to it
    - Publish and subscribe

1. **Event based** architectural style – publish subscribe is key.
   - **Event bus** – mechanism which the publishers and subscribers are matched → what coordinates these events
2. **Shared data space** architectural style – there's a database which is persistent and liable
   - The components will communicate entirely through **tuples** which is saved in a saved db, and other one does a quick search to see if the tuple exists any tuple that matches is returned.
   - **Tuples**: a structured data records with number of fields - Can be combined w event based – process subscribes to certain tuples ◊publish \&subscribe