Parallel & Distributed System

Distributed Systems Architecture-02

Md. Biplob Hosen

Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

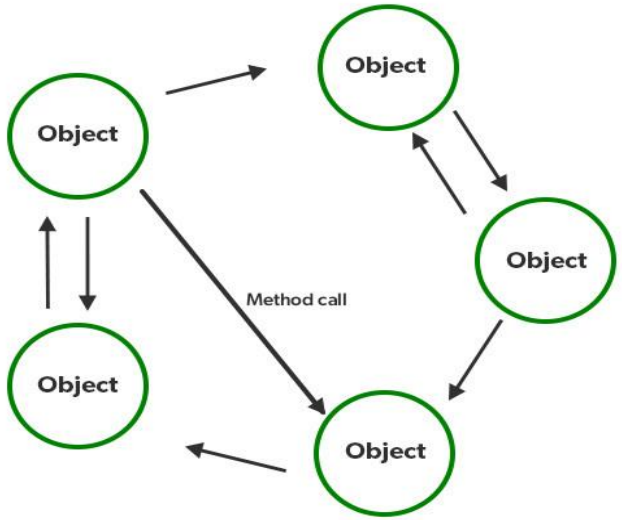
Contents

- Object-based Architectures
- Service-oriented Architectures
- Resource-based Architectures
- Publish-subscribe Architectures
- System Architecture

- Reference Books
  - Distributed Systems: Principles and Paradigms, 3rd Edition by Andrew S. Tanenbaum & Maarten van Steen, Publisher: Pearson Prentice Hall.

Object-based Architectures

- Components are objects, connected to each other through procedure calls. Objects may be placed on different machines; calls can thus execute across a network.
- Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.
- The interface offered by an object conceals implementation details, essentially meaning that we, in principle, can consider an object completely independent of its environment.



Object-based Architectures

- This organization, which is shown in following figure is commonly referred to as a distributed object, or sometimes a remote object.

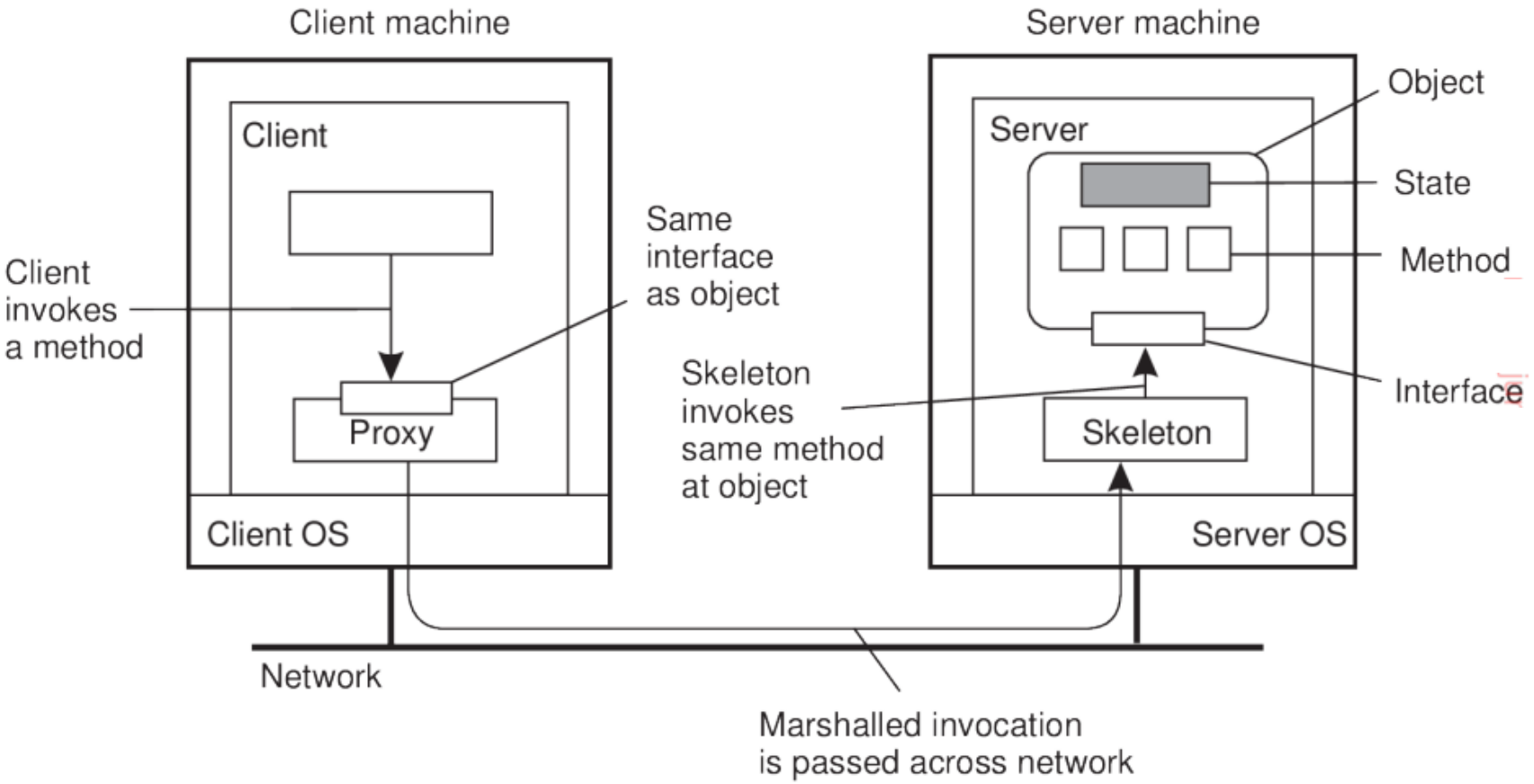


Figure 2.6: Common organization of a remote object with client-side proxy.

Object-based Architectures

- When a client binds to a distributed object, an implementation of the object's interface, called a proxy, is then loaded into the client's address space.
- A proxy is analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client.
- The actual object resides at a server machine, where it offers the same interface as it does on the client machine.
- Incoming invocation requests are first passed to a server stub, which unmarshals them to make method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.
- The server-side stub is often referred to as a skeleton as it provides the bare means for letting the server middleware access the user-defined objects.
- In practice, it often contains incomplete code in the form of a language-specific class that needs to be further specialized by the developer.

Service-based Architectures

- Object-based architectures form the foundation of encapsulating services into independent units.
- The service as a whole is realized as a self-contained entity, although it can possibly make use of other services.
- By clearly separating various services such that they can operate independently, we are paving the road toward **service-oriented architectures**, generally abbreviated as SOAs.
- In a service-oriented architecture, a distributed application or system is essentially constructed as a composition of many different services.
- Not all of these services may belong to the same administrative organization.
- For example, an organization running its business application makes use of storage services offered by a cloud provider. These storage services are logically completely encapsulated into a single unit, of which an interface is made available to customers.

Service-based Architectures

- Another example, a Web shop selling goods such as e-books.
- A simple implementation following the application layering may consist of an **application for processing orders**, which, in turn, operates on a local database containing the e-books.
- Order processing typically involves selecting items, registering and checking the delivery channel, but also making sure that a.
- The latter can be handled by a separate service, **payment takes place** run by a different organization, to which a purchasing customer is redirected for the payment, after which the e-book organization is notified so that it can complete the transaction.
- In this way, we see that the problem of developing a distributed system is partly one of service composition, and making sure that those services operate in harmony.
- Here, each service offers a well-defined (programming) interface. In practice, this also means that each service offers its own interface, in turn, possibly making the composition of services far from trivial.

Resource-based Architectures

- As an increasing number of services became available over the Web and the development of distributed systems through service composition became more important, researchers started to rethink the architecture of mostly Web-based distributed systems.
- One of the problems with service composition is that connecting various components can easily turn into an integration nightmare.
- As an alternative, one can also view a distributed system as a huge collection of resources that are individually managed by components.
- Resources may be added or removed by (remote) applications, and likewise can be retrieved or modified.
- This approach has now been widely adopted for the Web and is known as Representational State Transfer (REST).
- There are four key characteristics of what are known as RESTful architectures:

Resource-based Architectures

1. Resources are identified through a single naming scheme.
2. All services offer the same interface, consisting of at most four operations.
3. Messages sent to or from a service are fully self-described.
4. After executing an operation at a service, that component forgets everything about the caller.

Operation	Description
PUT	Modify a resource by transferring a new state
POST	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource

Figure 2.7: The four operations available in RESTful architectures.

Resource-based Architectures

- To illustrate how RESTful can work in practice, consider a cloud storage service, such as Amazon's Simple Storage Service (Amazon S3).
- Amazon S3 supports only two resources: objects, which are essentially the equivalent of files, and buckets, the equivalent of directories.
- There is no concept of placing buckets into buckets.
- An object named ObjectName contained in bucket BucketName is referred to by means of the following uniform resource identifier (URI):
  - http://s3.amazonaws.com/BucketName/ObjectName
- To create a bucket, or an object for that matter, an application would essentially send a PUT request with the URI of the bucket/object.
- In it is just a HTTP request, which will subsequently be correctly interpreted by S3.
- If the bucket or object already exists, an HTTP error message is returned.
- In a similar fashion, to know which objects are contained in a bucket, an application would send a GET request with the URI of that bucket.
- S3 will return a list of object names, again as an ordinary HTTP response.

Resource-based Architectures

Amazon S3 SOAP interface:

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

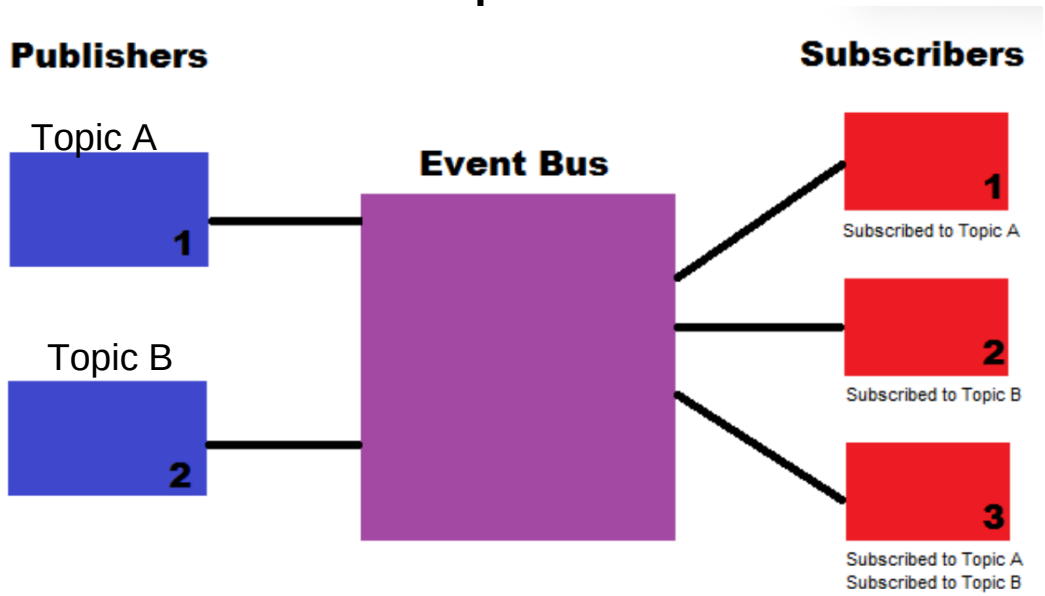
- The SOAP interface consists of approximately 16 operations. However, if we were to access Amazon S3 using the Python boto library, we would have close to 50 operations available. In contrast, the REST interface offers only very few operations.
- In the case of RESTful architectures, an application will need to provide all that it wants through the parameters it passes by one of the operations.
- In Amazon's SOAP interface, the number of parameters per operation is generally limited, and this is certainly the case if we were to use the Python boto library.
- Sticking to principles, suppose that we have an interface bucket that offers an operation create, requiring an input string such as mybucket, for creating a bucket with name "mybucket."
- Normally, the operation would be called roughly as follows:
  - import bucket
  - bucket.create("mybucket")
- However, in a RESTful architecture, the call would need to be essentially encoded as a single string, such as: PUT "http://mybucket.s3.amazonsws.com/"

Publish-subscribe Architectures

- There are three main components to the Publish Subscribe Model: publishers, eventbus/broker and subscribers.
- A PublishSubscribe Architecture is a messaging pattern where the publishers broadcast messages, with no knowledge of the subscribers.
- Similarly the subscribers ‘listen’ out for messages regarding topic/categories that they are interested in without any knowledge of who the publishers are.
- The event bus transfers the messages from the publishers to the subscribers.
- Each subscriber only receives a subset of the messages that have been sent by the publisher; they only receive the message topics or categories they have subscribed to.
- There are two methods of filtering out unrequired messages: topic-base filter or content-based filter.
- The topic-based filtering requires the messages to be broadcasted into logical channels, the subscribers only receives messages from logic channels they care about (and have subscribed to).
- A content-based system allows subscribers to receive messages based on the content of the messages and the subscribers themselves must sort out junk messages from the ones they want.

Publish-subscribe Architectures - Example

- In this topic-based example the Event Bus knows what topic each subscriber is subscribed to.
- The event bus will filter messages based on topic and send the messages to subscribers that are subscribed to the topic of the message.
- The publishers are responsible for defining the topics of their messages.
- In the above diagram, any message published with Topic A will be sent to Subscriber 1 and Subscriber 3.
- Similarly, any message published with Topic B will be sent to Subscriber 2 and Subscriber 3.



System Architecture

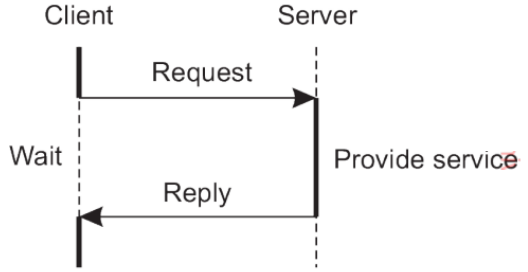
- Software components, their interaction, and their placement leads to an instance of a software architecture, also known as a **system architecture**.
- Centralized Organizations:
  - ✓ Simple client-server Architecture
  - ✓ Multitiered Architectures
- Decentralized Organizations:
  - ✓ Peer-to-peer Systems
- Hybrid Architectures
  - ✓ Edge-server Systems
  - ✓ Collaborative Distributed Systems

Simple client-server Architecture

- There are processes offering services (servers) & use services (clients).
- Clients follow request/reply model with respect to using services.
- Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks.
- In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary data.
- The message is then sent to the server, which, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.
- Using a connectionless protocol has the obvious advantage of being efficient.
- As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine.
- Unfortunately, making the protocol resistant to occasional transmission failures is not trivial.

Simple client-server Architecture

- As an alternative, many client-server systems use a reliable connection-oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly fine in wide-area systems in which communication is inherently unreliable.
- One of the main issues of the client-server model is how to draw a clear distinction between a client and a server.
- For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables.
- In such a case, the database server itself only processes the queries.

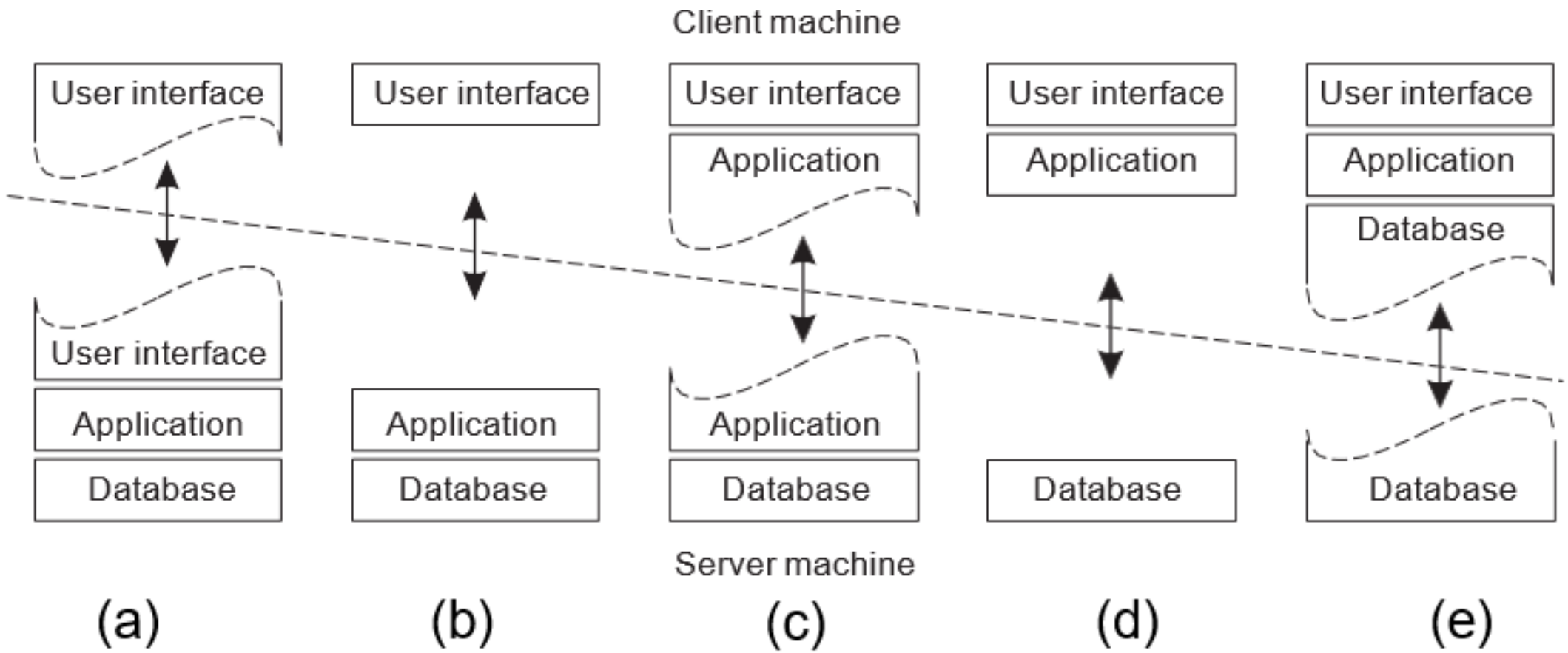


General interaction between a client and a server

Multitiered Architectures

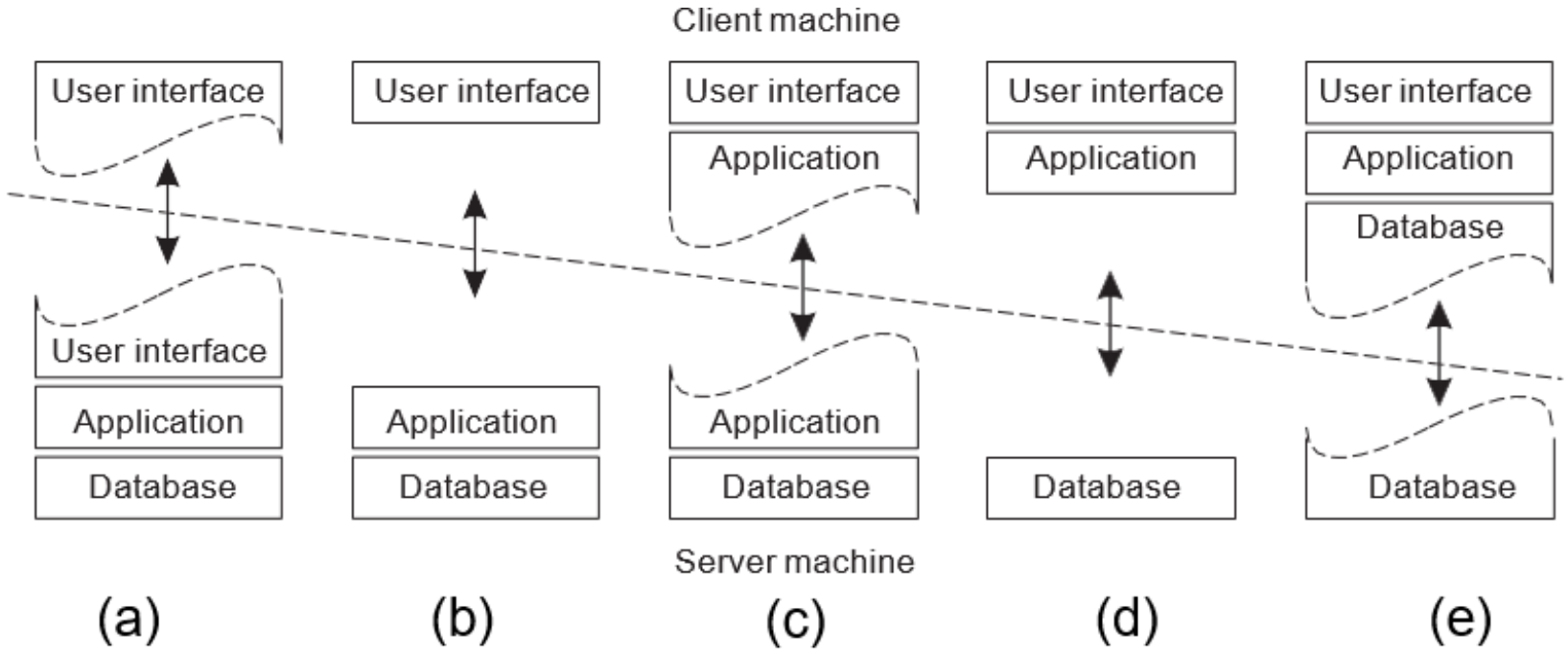
- The simplest organization is to have only two types of machines:
  1. A client machine containing only the programs implementing (part of) the user-interface level.
  2. A server machine containing the rest, that is, the programs implementing the processing and data level.
- In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with only a convenient graphical interface.
- There are, however, many other possibilities. As explained earlier, many distributed applications are divided into the three layers (1) user interface layer, (2) processing layer, and (3) data layer.
- One approach for organizing clients and servers is then to distribute these layers across different machines.
- As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two-tiered architecture.

Multitiered Architectures - 2 Tier



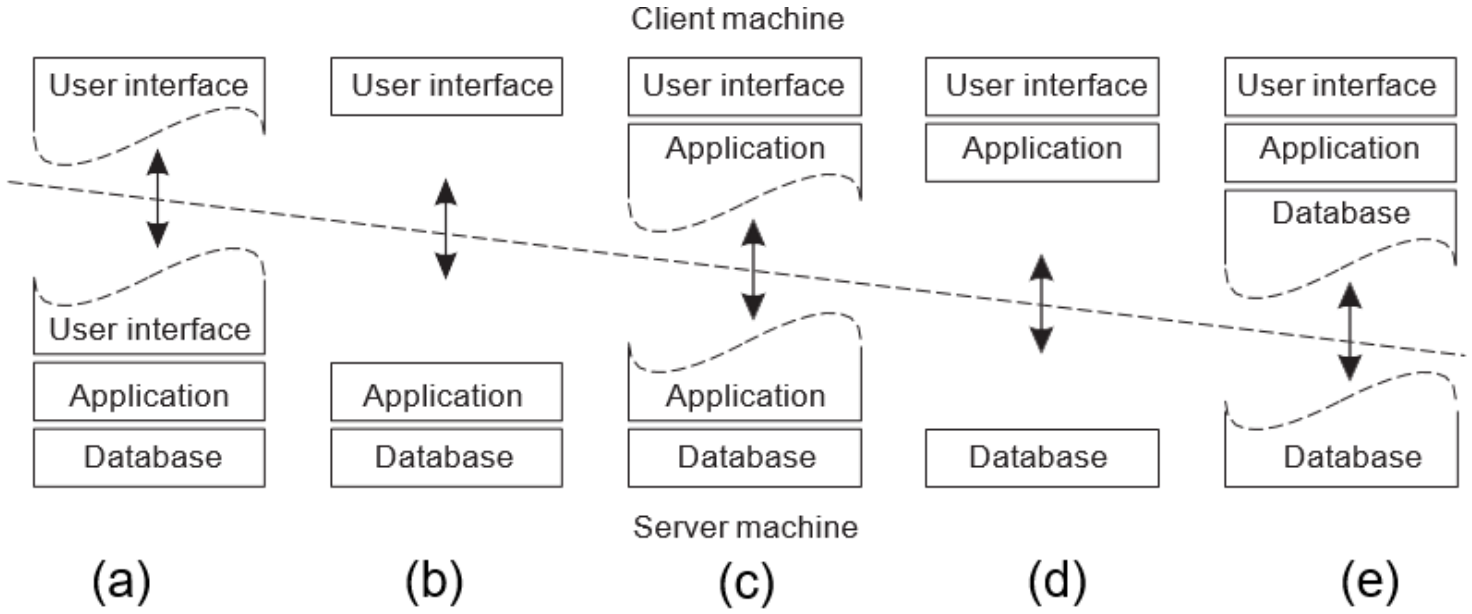
- One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in (a), and give the applications remote control over the presentation of their data.

Multitiered Architectures - 2 Tier



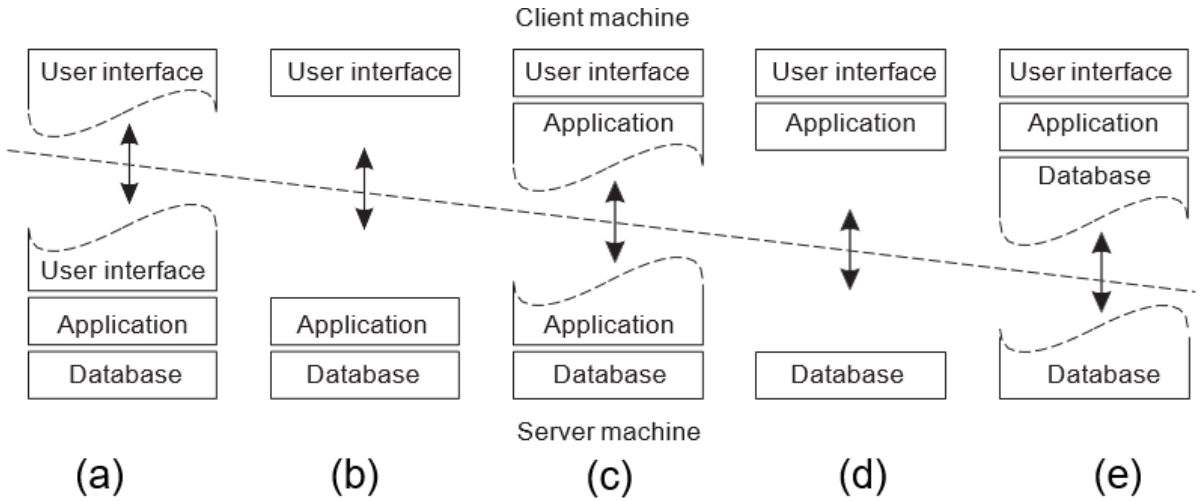
- An alternative is to place the entire user-interface software on the client side, as shown in (b).
- In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol.
- In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

Multitiered Architectures - 2 Tier



- Continuing along this line of reasoning, we may also move part of the application to the front end, as shown in (c).
- An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed.
- The front end can then check the correctness and consistency of the form, and where necessary interact with the user.
- Another example of this organization is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data, but where the advanced support tools such as checking the spelling and grammar execute on the server side.

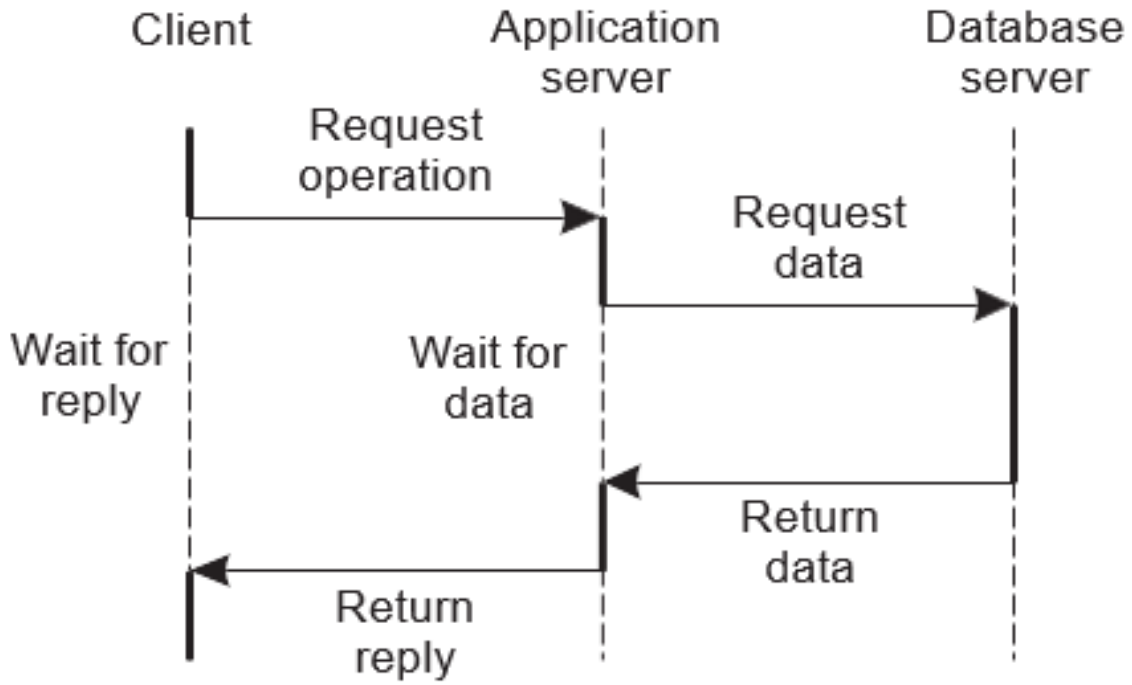
Multitiered Architectures - 2 Tier



- In many client-server environments, the organizations in (d) & (e) are particularly popular.
- These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database.
- Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server.
- For example, many banking applications run on an end-user's machine where the user prepares transactions and such.
- Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing.
- Figure (e) represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

Multitiered Architectures - 3 Tier

- When distinguishing only client and server machines as we did so far, we miss the point that a server may sometimes need to act as a client, leading to a three-tiered architecture.



Multitiered Architectures - 3 Tier

- In this architecture, traditionally programs that form part of the processing layer are executed by a separate server, but may additionally be partly distributed across the client and server machines.
- A typical example of where a three-tiered architecture is used is in transaction processing.
- A separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers.
- Another, but very different example where we often see a three-tiered architecture is in the organization of Web sites.
- In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place.
- This application server, in turn, interacts with a database server.
- For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore.
- To do so, it may need to interact with a database containing the raw inventory data.

Thank You ☺