Distributed Systems
ECEG-6504

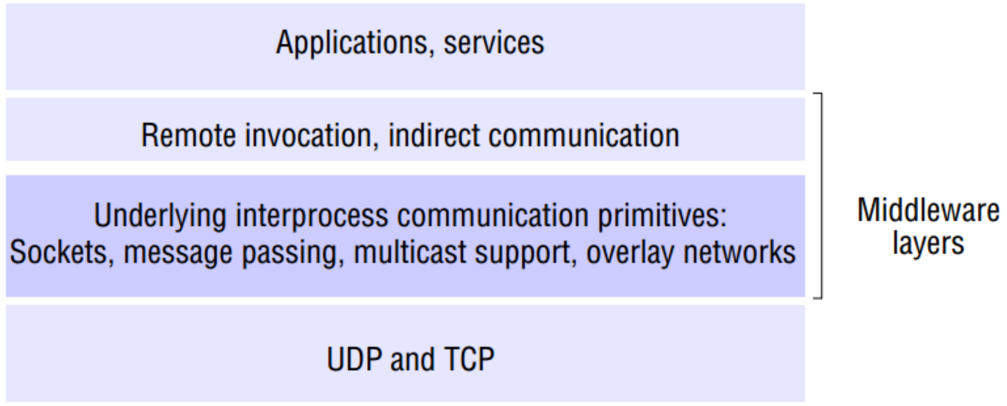
Distributed Objects and Remote Invocation

Surafel Lemma Abebe (Ph. D.)

Topics

- Introduction
- Remote procedure call
- Communication between distributed objects
- Events and notifications

Introduction


Applications, services
Remote invocation, indirect communication
Underlying interprocess communication primitives: Sockets, message passing, multicast support, overlay networks
UDP and TCP
Middleware layers

- Middleware
  - Software that provides a programming model above the basic building blocks of processes and message passing
  - Invented to provide common services and protocols that can be used by many different applications
  - Important aspect
    - Provide location transparency and independence from the details of communication protocols, operating systems, and computer hardware

Introduction…

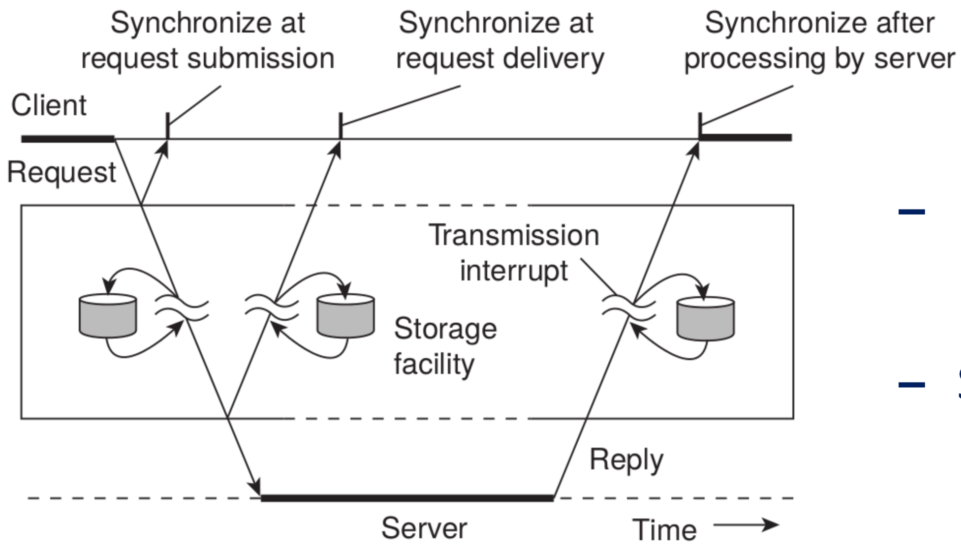
- Middleware…
  - Provide location transparency
    - RPC
      - Client that calls a procedure cannot tell whether the procedure runs in the same process or in a different process, nor does the client need to know the location of the server
    - RMI
      - The object making the invocation cannot tell whether the object from which it invokes a method is local or remote
    - Distributed event-based programs
      - Objects generating events and the objects that receive notifications of those events need not be aware of one another's location

Introduction…

- Middleware…
  - Provide independence from the details of
    - Communication protocols
      - The protocols that support the middleware abstractions are independent of the underlying transport protocols
    - Operating systems
      - The higher-level abstractions provided by the middleware layer are independent of the underlying operating systems
    - Computer hardware
      - Data representation details

Introduction…

- Communication types
  - Transient communication
    - Message is not stored by the middleware
    - Both sender and receiver should be up and running for the message to be delivered
    - Sender and receiver are coupled in time
  - Persistent communication
    - Message is stored
  - Synchronous communication
    - Message acknowledgement (ack)
    - Ack at request submission and delivery are controversial
    - Ack after processing by server is considered as synchronous by all
  - Asynchronous communication


Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Introduction...

- Concerned with how processes (or entities) at a higher level of abstraction (e.g. objects) communicate in a DS
  - Request-reply protocols
    - Represent a pattern on top of message passing
    - Supports a two-way exchange of messages as encountered in client-server computing
  - Remote procedure call (RPC) model
    - Extension of the conventional procedure call model to DS
    - Allows client programs to call procedures transparently in server programs running in separate processes
  - Remote method invocation (RMI)
    - Extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process
  - Indirect communication

Remote procedure call

- Makes the programming of DS look similar to conventional programming
  - Achieve high level of distribution transparency
- Procedures on remote machines can be called as if they are procedures in the local address space
- Birrell and Nelson [1984]
- Underlying RPC system hides
  - Encoding and decoding of parameters and results
  - Passing of messages and preserving of the required semantics for the procedure call

Remote procedure call…

- Design issues for RPC
  - Issues that are important in understanding implementation of RPC systems
    - Programming with interfaces (style of programming promoted by RPC)
    - Call semantics associated with RPC
    - Issues of transparency and how it relates to remote procedure calls

Remote procedure call…

- Design issues for RPC…
  - Programming with interfaces
    - Refers to the style of programming promoted by RPC
    - Interfaces are used to control the possible interactions between modules
    - Interface of a module specifies procedures and variables that can be accessed from other modules
    - Interfaces in DS
      - Modules can run in separate processes
      - Service interface
        - » Refers to the specification of the procedures offered by a server, defining the types of the arguments of each of the proc

Remote procedure call…

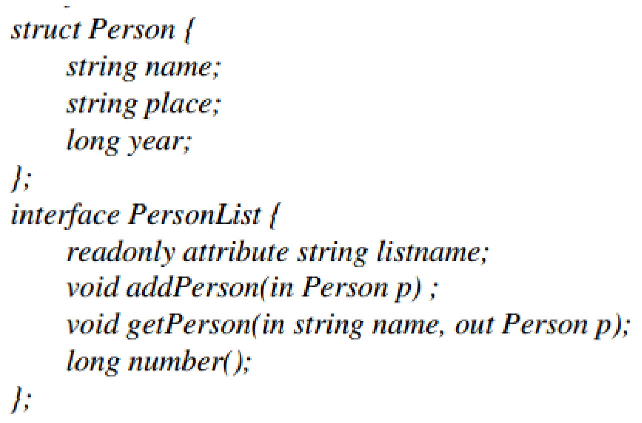
- Design issues in RPC…
  - Programming with interfaces…
    - Advantages of programming with interfaces in DS
      - Programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details
      - Programmers do not need to know the programming language or underlying platform used to implement the service
      - Provides natural support for software evolution
        - » Implementations can change as long as the interface remains the same
        - » Interface can also change as long as it remains compatible with the original

Remote procedure call...

- Design issues in RPC...
  - Programming with interfaces...
    - Distribution influence on definition of service interfaces
      - Service interface cannot specify direct access to variables
        - » Its not possible for a client module running in one process to access the variables in a module in another process
      - Parameter-passing mechanisms used in local procedure calls are not suitable
        - » Call by reference is not supported
        - » Specification describes the parameters as input and output, or both
          - Sends the values
      - Addresses in one process are not valid in another remote process
        - » Addresses cant be passed as arguments or returned as results of calls to remote modules

Remote procedure call…

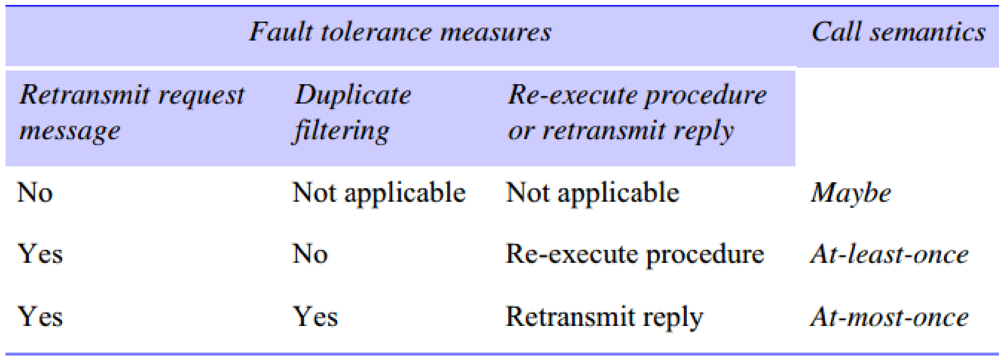
- Design issues in RPC…
  - Programming with interfaces…
    - Interface definition languages (IDL)
      - Designed to allow procedures implemented in different languages to invoke one another
      - Provides a notation for defining interfaces
        - » Used to describe input or output parameters of an operation and corresponding types
      - Example: IDL for *struct person*



```
struct Person {
        string name;
        string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

Remote procedure call…

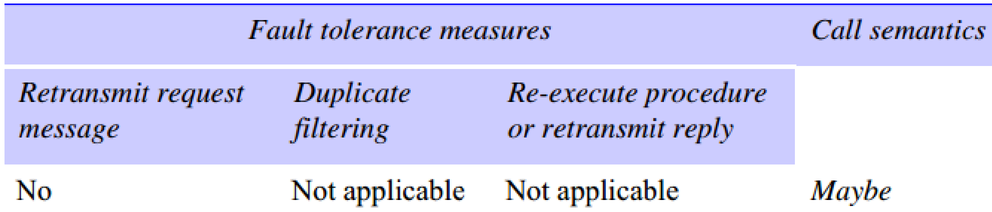
- Design issues in RPC…
  - RPC call semantics
    - Recap: doOperation implementation choices
      - Retry request message
      - Duplicate filtering
      - Retransmission of results
    - Combination of these choices leads to the following possible semantics for the reliability of remote invocations as seen by the invoker



Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Remote procedure call…

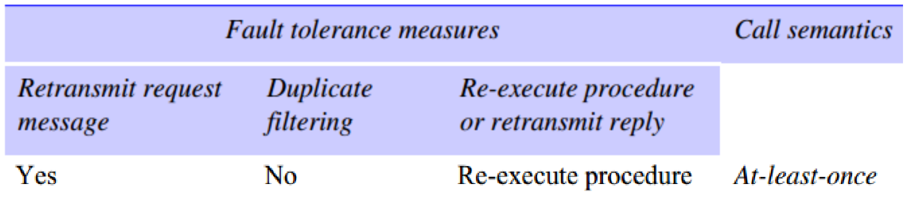
- Design issues in RPC…
  - RPC call semantics…
    - *Maybe* semantics
      - Remote procedure call may be executed once or not at all
      - No fault tolerance measures are applied
      - Suffers from the following types of failures
        - » Omission failures, if the request or result message is lost
        - » Crash failures when the server containing the remote operation fails
      - In an asynchronous system
        - » The result may arrive after the timeout
      - Useful for applications in which occasional failed calls are acceptable



Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*

Remote procedure call…

- Design issues in RPC…
  - RPC call semantics…
    - At-least-once semantics

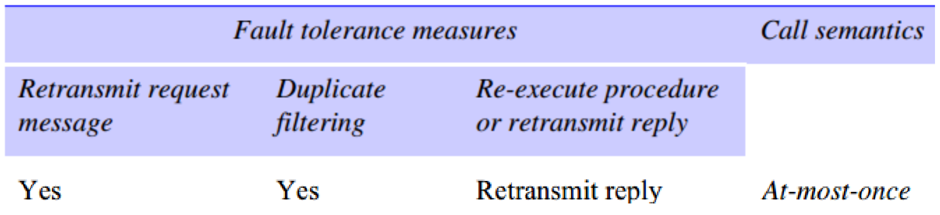


Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
Yes	No	Re-execute procedure	*At-least-once*

      - Invoker receives either a result or an exception
      - Masks the omission failures of the request or result message
      - Suffers from the following types of failures
        - » Crash failures when the server containing the remote procedure fails
        - » Arbitrary failures – due to re-execution of a procedure which causes wrong values to be stored or returned
      - Acceptable if the operations in the server are idempotent

Remote procedure call…

- Design issues in RPC…
  - RPC call semantics…
    - At-most-once semantics
      - The caller receives either a result or an exception
        - » A procedure is executed at most once



Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
Yes	Yes	Retransmit reply	*At-most-once*

Remote procedure call…

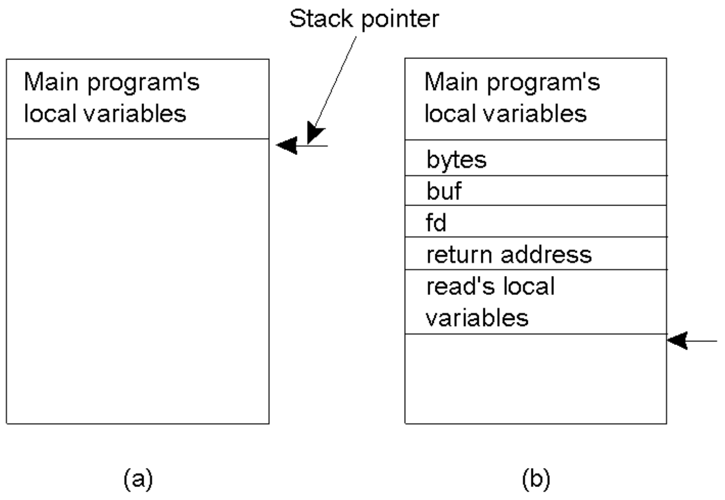
- Design issues in RPC…
  - Transparency
    - Aim is to make RPC like local procedure call, with no distinction in syntax
      - All calls to marshalling and message-passing procedures are hidden from the programmer making the call
      - Re-sending request is transparent to the caller
    - Strives to offer location and access transparency
    - RPCs are vulnerable to failure than local calls
      - Difficult to distinguish between failure of the network and of the remote server process
      - Clients making remote calls are required to recover from such failures
    - There is a higher latency for RPC

Remote procedure call…

- Design issues in RPC…
  - Transparency…
    - Suggestions
      - Argus designers suggested that a caller should be able to abort a remote procedure call that is taking too long in such a way that it has no effect on the server [Liskov and Scheifler 1982]
      - Difference between local and remote operations should be expressed at the service interface, to allow participants to react in a consistent way to possible partial failures [Waldoet al.1994]
      - Syntax of a remote call should be different from that of a local call
    - Consensus
      - Syntax of remote call is the same as that of a local invocation, but that the difference between local and remote calls should be expressed in their interfaces

Remote procedure call…

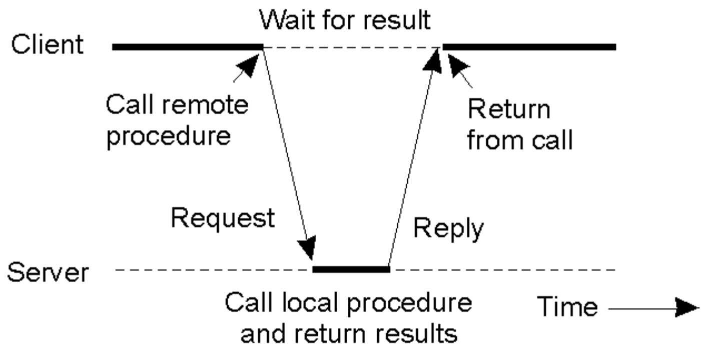
- Basic RPC operation
  - Conventional procedural call
    - Example: count=read(fd, buf, bytes)
    - Steps to call a procedure
      1. Caller pushes parameters onto the stack
      2. Push return address on stack
      3. Run read operation
      4. Upon completion of read operation, put the result in a register, remove the return address and transfer control back to the caller
      5. Caller removes the parameters from the stack (return the stack to the original state)



a) The stack before the call to read
b) The stack while the called procedure is active

Remote procedure call…

- Basic RPC operation…
  - Client and server stubs
    - Calling a remote procedure is similar to calling a local procedure
    - Client stub
      - Similar to functions such as read
      - Makes a call to the local OS
      - Difference from a non-RPC calls
        - » It packs the parameters into a message and requests the message to be sent to the server
        - » After call to send, it blocks itself until the reply comes back
        - » When a reply comes, it inspects the message, unpacks the result, and copies it to the caller

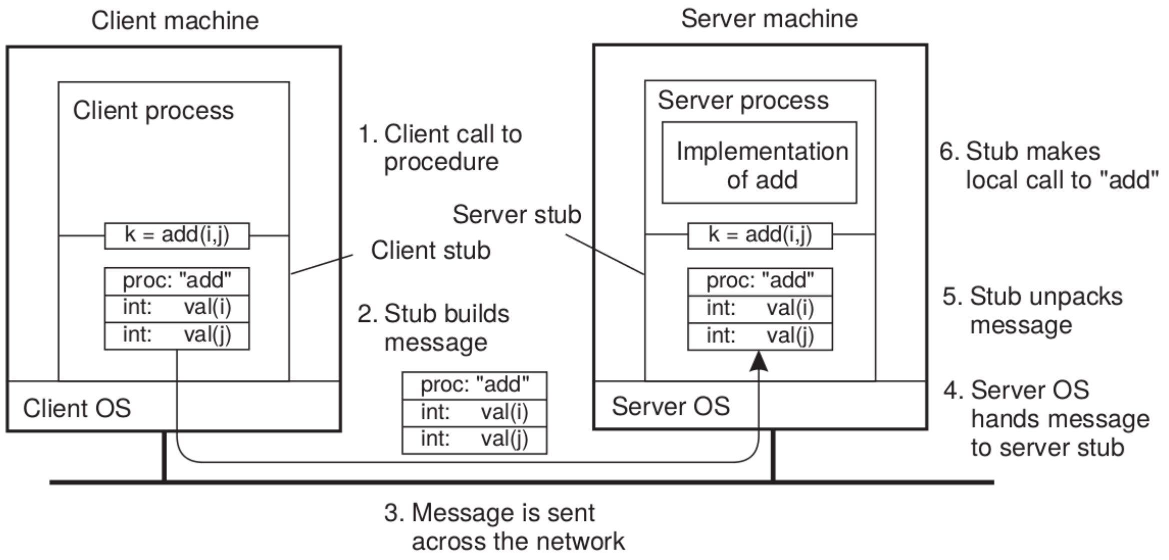


Remote procedure call…

- Basic RPC operation…
  - Client and server stub…
    - Server stub
      - Receives messages from the server OS
      - Blocks waiting for incoming messages
      - Transforms requests coming in over the network into local procedure calls
        - » Unpacks the parameters from the message
        - » Calls the server procedure in the usual way
      - When it gets control back after the call has completed
        - » It packs the result in a message
        - » Calls send to return it to the client

Remote procedure call…

- Basic RPC operation…



1. Client procedure calls client stub
2. Stub builds a message, calls local OS
3. Client OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters and calls the server
6. Server returns result to stub
7. Server stub packs the message; calls OS
8. OS sends message to client’s OS
9. Client OS gives message to client stub
10. Client stub unpacks result and returns to the client

Remote procedure call…

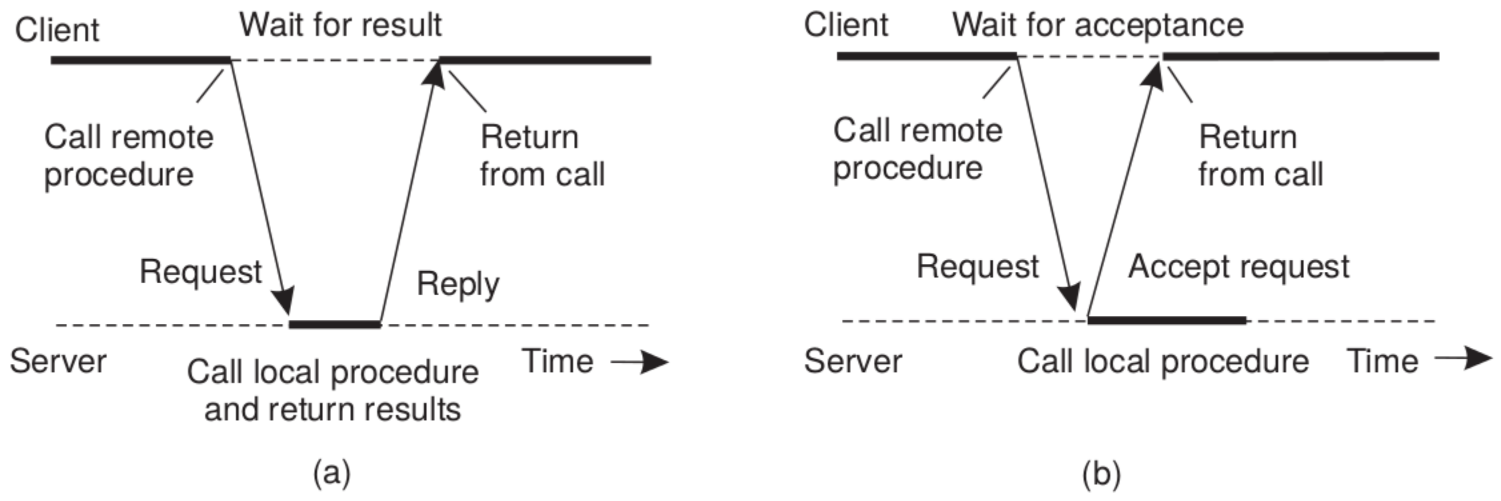
- Parameter passing
  - By value
  - By reference
    - Difficult to achieve. Why?
    - Solutions
      1. Forbid pointers and reference parameters
      2. Copy/restore
         - » Copy the variable as in call-by-value, and then copy back after the call, i.e., overwrite the caller's original value

Remote procedure call…

- Asynchronous RPC
  - When there is no result to return, the strict request-reply behavior is not necessary
  - Example
    - Adding entries in a DB, starting remote services, …
  - Server immediately sends a reply (ack) when it receives an RPC request

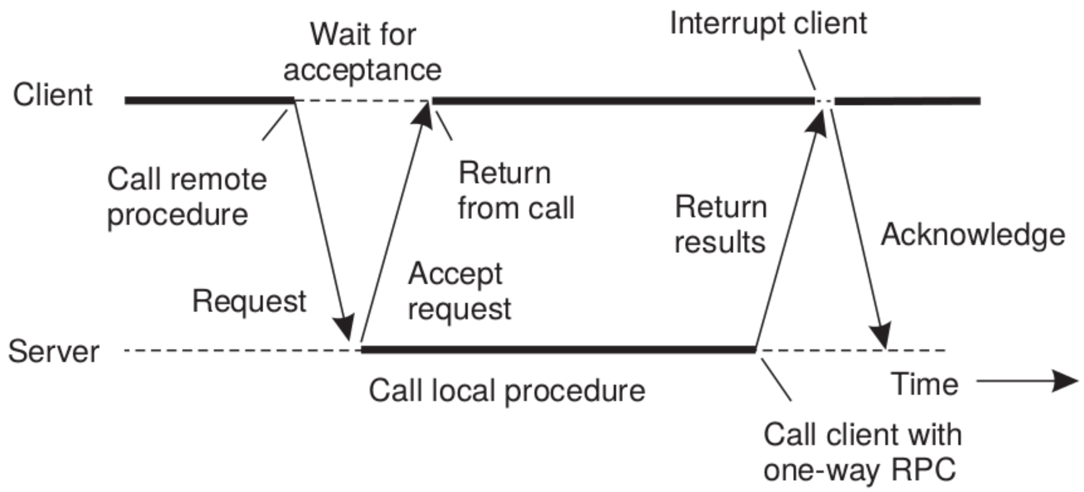
(a) Traditional RPC

(b) Asynchronous RPC



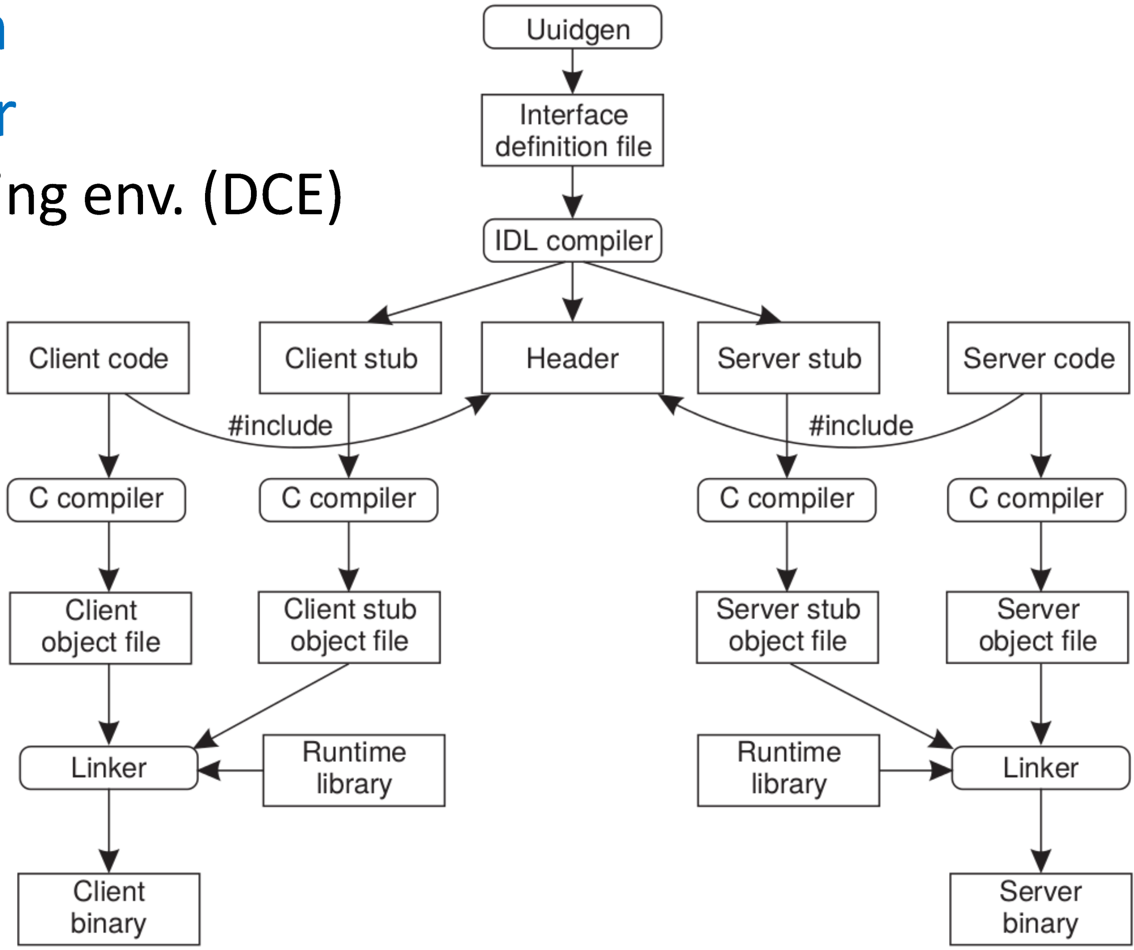
Remote procedure call…

- Deferred synchronous RPC
  - Useful when a reply will be returned but the client is not prepared to wait for it
  - Implemented using two asynchronous RPCs
  - First asynchronous RPC
    - The client submits the job and continues after it gets an ack
  - Second asynchronous RPC
    - Server calls the client to hand over the result
    - Client can also do a (non)blocking poll at the server to see whether results are available



Remote procedure call…

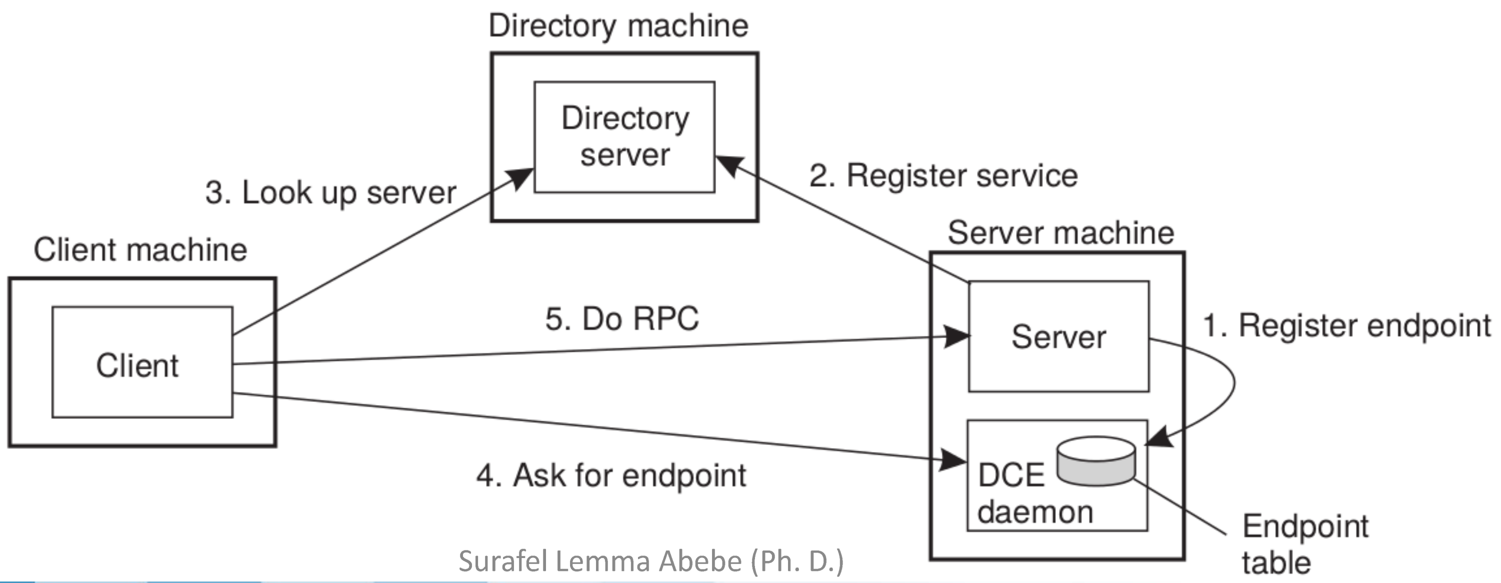
- Process of writing an RPC client and server
  - Distributed computing env. (DCE)
  - Uuidgen
    - Generates a prototype IDL file



Remote procedure call…

- Binding a client to a server in DCE
  - To allow a client to call a server, the server needs to be registered and prepared to accept calls
  - Locating a server is done in two steps
    - Locate the server's machine
    - Locate the server (i.e., the correct process) on the machine

      => The client needs to know the end point on the server machine



Communication between distributed objects

- Communication between distributed objects is addressed by means of RMI
- RMI (Remote Method Invocation)
  - Closely related to RPC but extended into the world of distributed objects
  - A calling object can invoke a method in a potentially remote object
  - Underlying details are hidden from the user

- Communalities between RMI and RPC
  - Both support programming with interfaces
  - Both are constructed on top of request-reply protocols
  - Both can offer a range of call semantics such as at-least-once
  - Both offer similar level of transparency
    - i.e., local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call

Communication between distributed objects…

- Differences between RPC and RMI
  - Programmer is able to use the full expressive power and related methodologies of object-oriented programming
  - All objects have unique object references, which can be passed as parameters
  - RMI also allows passing parameters by object reference

Communication between distributed objects…

- Design issues for RMI
  - RMI shares the same design issues as RPC in terms of
    - Programming with interfaces
    - Call semantics
    - Level of transparency
  - Added design issues are related to
    - Object model
    - Transition from objects to distributed objects

Communication between distributed objects…

- Object model
  - key feature of an object
    - Encapsulates data, called the state
    - Operations on the data, called the methods
  - Some OO languages allow object’s data to be accessed directly
  - In distributed object system, an object’s data should be accessible only via its methods

  - Object reference
    - Used to access an object
    - Could be assigned to variables, passed as arguments, …
    - Object reference and the method name are used to invoke a method
  - Interfaces
    - Provides a definition of the signatures of a set of methods
    - A class may implement several interfaces, and the methods of an interface may be implemented by any class

Communication between distributed objects…

- Object model…
  - Actions
    - Initiated by an object invoking a method in another object
    - Effects of invoking a method
      - State of the receiver may be changed
      - A new object may be instantiated
      - An invocation could invoke a method in another object
  - Exceptions
    - Provide a clean way to deal with errors and unexpected events without complicating the code
    - A block of code could ***throw*** an exception

      => control passes to another block of code that ***catches*** the exception

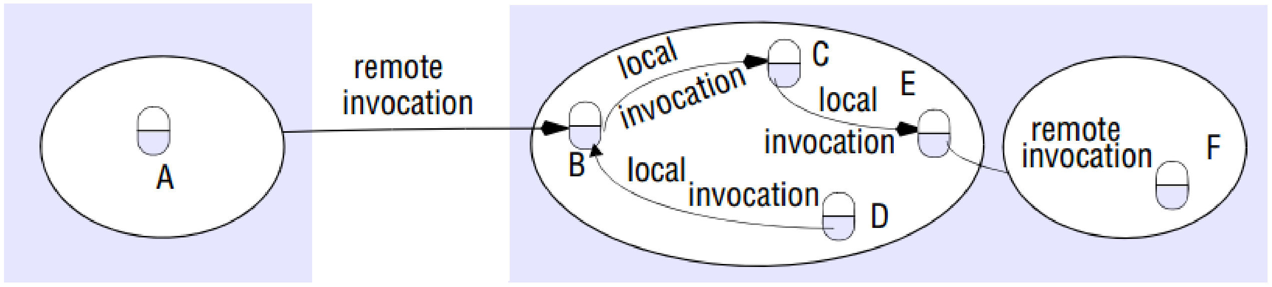
      NB. Control doesn't return to the place where the exception was thrown
  - Garbage collection
    - Used to free a memory space used by objects that are no longer used
    - If the language doesn't support it, the programmer has to deal with it
      - Example: C++

Communication between distributed objects…

- Distributed objects
  - Object-based programs are logically partitioned which supports physical distribution of objects
  - Usually adopt client-server architecture
    - Methods of objects on servers are invoked using RMI
      - Supports heterogeneity
    - Steps
      - Clients send the RMI request in a message to a server
      - The server executes the invoked method of the object
      - The server returns the result to the client in another message
  - Other architectural models
    - Objects in servers could also become clients of objects in other servers
    - Objects can be replicated
    - Objects can be migrated
  - Having client and server objects in different processes enforces encapsulation
    - The state of an object can be accessed only by the methods of the object
    - Objects can protect their instance variables from concurrent access

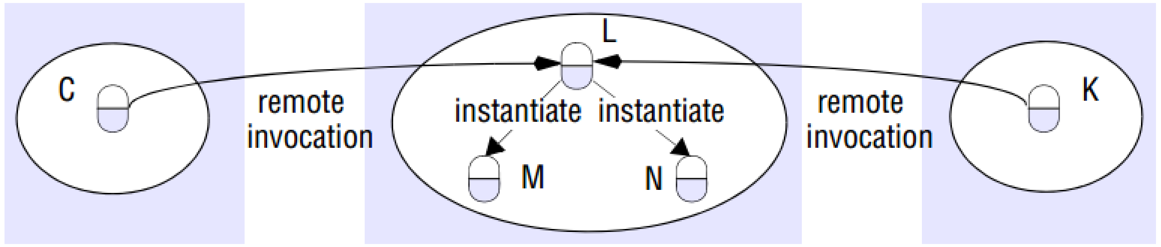
Communication between distributed objects…

- Distributed object models…
  - Terminologies
    - Remote Method Invocation
      - Method invocations between objects in different processes (e.g., B and F)
    - Local Method Invocations
      - Method invocations between objects in the same process (e.g., C and D)
    - Remote objects
      - Objects that can receive remote invocations



Communication between distributed objects…

- Distributed object models…
  - Actions
    - Initiated by a method invocation
    - Remote invocations could lead to the instantiation of new objects
  - Exceptions
    - RMI should be able to raise additional exceptions that are related to distribution
      - E.g., timeout
  - Distributed garbage collection
    - Achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection
    - Its usually based on reference counting

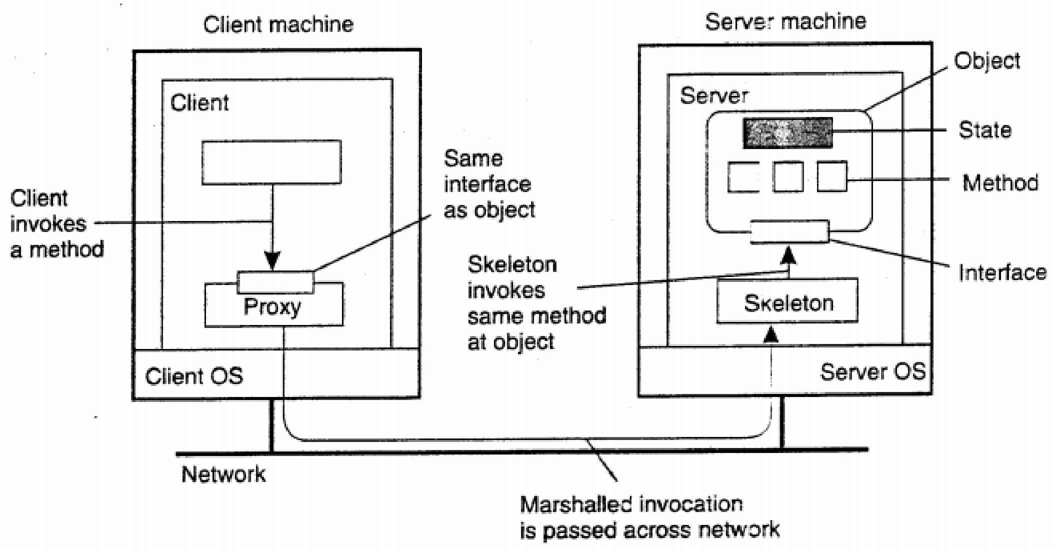


Communication between distributed objects…

- Distributed object models…
  - Fundamental concepts
    - Remote object references
      - Extends object references
      - Is an identifier that can be used throughout a DS to refer to a particular unique remote object
      - Other objects can invoke the methods of a remote object if they have access to its remote object reference
      - Can be passed as arguments
    - Remote interfaces
      - Used by remote objects to specify which of its methods can be invoked remotely
      - IDL is used to define remote interfaces
      - An interface may be implemented by several objects
      - Separation of interfaces and the object implementation allows to place an interface at one machine, while the object itself could reside on another machine

Communication between distributed objects…

- Distributed objects…
  - When a client binds to a distributed object, an implementation of the objects interface, called **proxy**, is loaded into the client's address space
  - Skeleton
    - Is the server stub
    - Contains incomplete code in the form of a language specific class that needs to be further specialized by the developer
  - State of the object is not distributed or its hidden from the developer



Communication between distributed objects…

- Distributed object models…
  - Object forms in DS
    - Compile-time object
      - Directly related to language-level objects
      - An object is defined as an instance of a class
      - Makes development of distributed applications easier
        - » Compiling class definition results in code that allows to instantiate objects
        - » An interface could be compiled into client-side and server-side stubs
        - » Developer is unaware of the distribution of objects, but sees only the code
      - Drawback
        - » Dependency on programming language

Communication between distributed objects…

- Distributed object models…
  - Object forms in DS…
    - Runtime objects
      - Objects are constructed at runtime
      - Its independent of the programming language in which the distributed application is written
      - How objects are actually implemented is left open
        - » E.g., C library containing a number of functions
        - » How the implementation appear to be an object depends on the approach used
        - » Common approach used is object adapter
          - E.g., dynamically binds to a C library and opens an associated data file representing an object’s current state

Communication between distributed objects…

- Distributed object models…
  - Object forms in DS…
    - Persistent object
      - Continues to exist even if it is not currently contained in the address space of any server process
      - The object’s state could be stored on a secondary storage
    - Transient object
      - Exists as long as the server that is hosting the object exists
- Reading assignment
  - Implementation of RMI

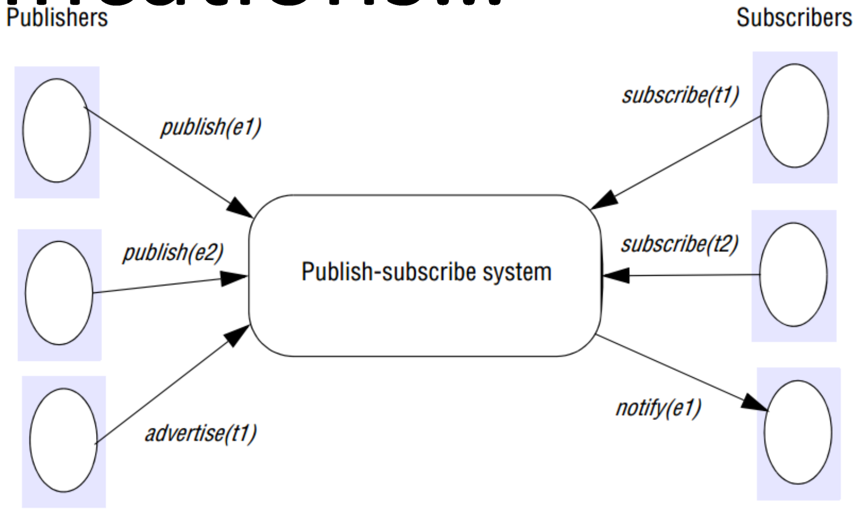
Events and notifications

- Idea behind the use of events
  - One object can react to a change occurring in another object
- Example
  - Interactive application: actions performed on objects
- Notification of events are asynchronous and determined by their receivers
- Distributed event-based systems
  - Extend the local event model
  - Allow multiple objects at different locations to be notified of events taking place at an object
  - Use publish-subscribe paradigm
    - Publishers publish structured events to an event service
    - Subscribers express interest in particular events through subscriptions

Events and notifications…

- Publish-subscribe…
  - Publish-subscribe system
    - » Matches subscriptions against published events
    - » Ensures the correct delivery of event notification
  - Is a one-to-many communication paradigm
    - Event is delivered to potentially many subscribers
  - Main characteristics
    - Heterogeneity
      - Components in a DS that were not designed to interoperate can be made to work together
      - A way to standardize communication
    - Asynchronicity
      - Notifications are sent asynchronously by event-generating publishers
      - Publishers and subscribers are decoupled
      - No need for a publisher to wait for each subscriber, subscribers come and go

Events and notifications…



- Programming model
  - Operations
    - publish(e)
      - Publishers disseminate an event, e
    - subscribe(f)
      - f = filter, a pattern defined over a set of all possible events
      - Subscribers express interest in a set of events through subscription
      - Expressiveness of a filter is determined by the subscription model
    - notify(e)
      - Used to deliver events
    - unsubscribe(f)
      - Subscribers could revoke their interest
    - advertise(f)
      - Used to declare the nature of future events
      - Defined in terms of events of interest
    - unadvertise(f)
      - Used to revoke advertisements

Events and notifications…

- Programming model…
  - Subscription models
    - Channel-based
      - Publishers publish events to named channels and subscribers subscribe to one of these named channels to receive all events sent to that channel
      - Primitive scheme and the only one that defines a physical channel
      - Other schemes employ some form of filtering over the content
    - Topic-based (subject-based)
      - Assumption: each notification is expressed in terms of a number of fields, with one field denoting the topic
        - » Topics are explicitly declared here (vs. implicit definition in channel-base)
      - Subscriptions are defined in terms of the topic of interest
      - Can be enhanced using hierarchical organization of topics
        - » E.g., topic X and topic X/Y, subscribers of Y will be notified only for Y

Events and notifications…

- Programming model…
  - Subscription models…
    - Content-based
      - Generalization of topic-based approaches
      - Allows expression of subscription over a range of fields in an event notification
      - Is a query defined in terms of composition of constraints over the values of event attributes
        - » E.g., event related to the topic X, where it has A and B as its field values
    - Type-based
      - Linked with object-based approaches
      - Subscriptions are defined in terms of types of events
      - Matching is defined in terms of types or subtypes of the given filter
      - Advantage:
        - » Could be integrated in programming language
        - » Could check type correctness of subscriptions

Events and notifications…

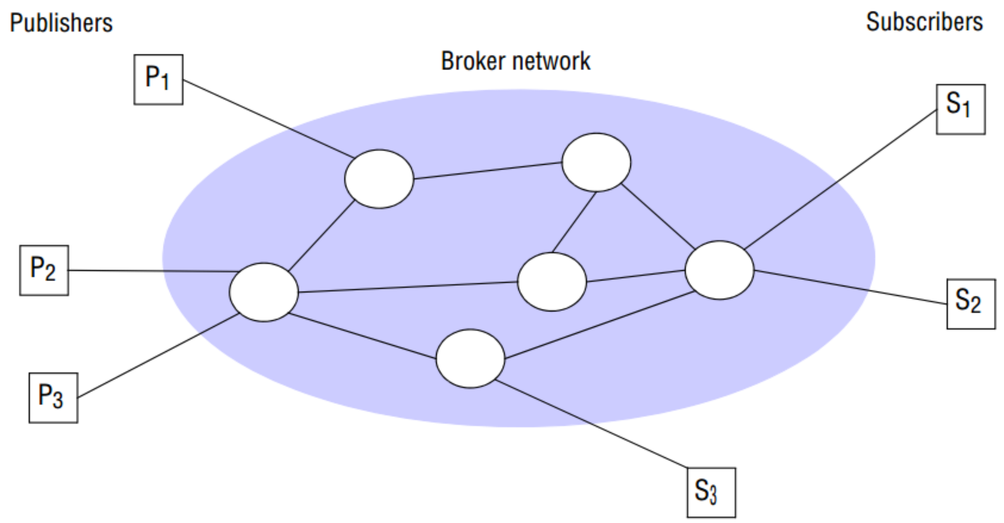
- Programming model…
  - Subscription models…
    - Example of experimental models
      - Context-based subscription
        - » Context and context awareness in mobile and ubiquitous computing
        - » Context
          - Aspect of the physical circumstances of relevance to the system behavior
          - E.g., location
      - Concept-based subscription
        - » Filters are expressed in terms of semantics and syntax of events

Events and notifications…

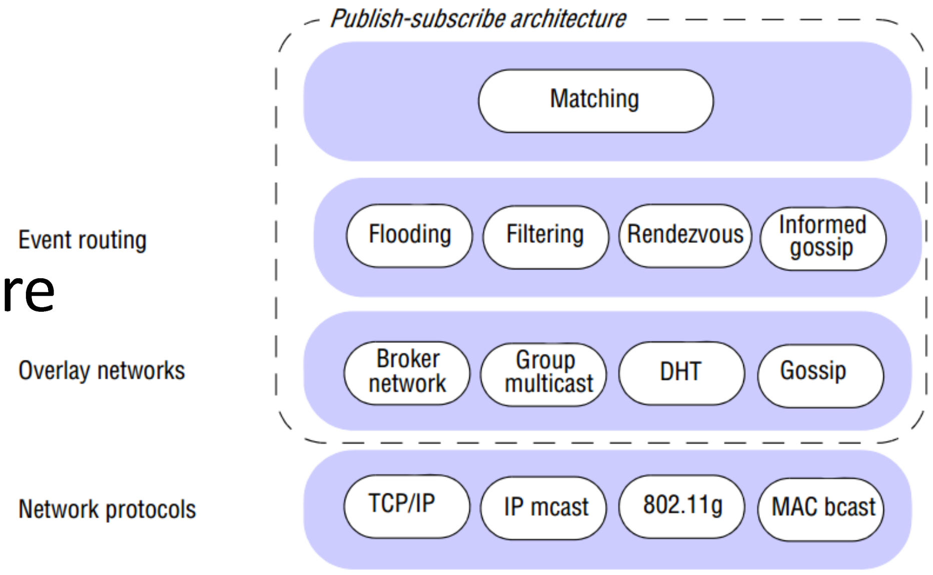
- Implementation issues
  - Centralized implementation
    - Simplest approach
    - Central server acts as an event broker
    - Publishers publish events and optionally send advertisements to the broker
    - Subscribers send subscriptions to the broker and receive notifications in return
    - Interaction with the broker is through a series of point to point messages
    - Disadvantage
      - Lacks resilience and scalability
      - Centralized broker
        - » Single point for potential system failure
        - » Performance bottleneck

Events and notifications…

- Implementation issues…
  - Distributed implementation
    - Centralized broker is replaced by a network of brokers
    - Has the potential to survive node failure



Events and notifications…



- Implementation issues…
  - Overall system architecture
    - Network protocols
      - Communication services
    - Top layer
      - Implements matching
      - Ensures that events match a given subscription
    - Event routing
      - Performs the task of ensuring that event notifications are routed as efficiently as possible to appropriate subscribers
      - For content-based approaches, its called content based routing (CBR)
    - Overlay networks
      - Used to setup appropriate networks of brokers or peer-to-peer structures

Events and notifications…

- Implementation issues…
  - Principles behind content-based routing
    - Flooding
      - Sends an event notification to all nodes in the network and then carries out the appropriate matching at the subscriber end
      - Could be used to send subscriptions back to all possible publishers with the matching carried out at the publishing end
        - » Matched events are sent directly to the relevant subscribers using point-to-point communication
      - Could be implemented
        - » Using an underlying broadcast or multicast facility or
        - » By arranging brokers in an acyclic graph in which each forwards incoming event notification to all its neighbors
      - Disadvantage
        - » Could result in a lot of unnecessary network traffic

Events and notifications…

- Implementation issues…
  - Principles behind content-based routing…
    - Filtering
      - Applies filtering in the network brokers
        - » Filtering-based routing
      - Brokers forward notifications through the network only where there is a path to a valid subscriber
        - » Achieved by propagating subscription information through the network
      - Each node maintains a neighbors list containing
        - » List of all connected neighbors in the network of brokers
        - » A subscription list containing a list of all directly connected subscribers serviced by this node
        - » A routing table, which maintains a list of neighbors and valid subscriptions for that pathway
      - Implements a matching function that returns a set of nodes where a notification matches the subscription

Events and notifications…

- Implementation issues…
  - Principles behind content-based routing…
    - Gossip-based
      - Operate by nodes in the network periodically and probabilistically exchanging events (or data) with neighboring nodes
      - Pure gossip-based approach is an alternative strategy for implementing flooding