Chapter0. Introduction

2019年2月20日 8:01

Note Taking Area

Key textbook

- Distributed Systems: Concepts and Design: 分布式系统概念与设计
- Mastering Cloud Computing

Course information

- BOOKS: Using DSCD for about 10 weeks, and the last 2 or 3 weeks use MCC.
- Grade percentage:
  - Final exam 70% - let me die!
  - Lab and Assignment 30%.
- Don't miss labs!

Cue Column

Summaries

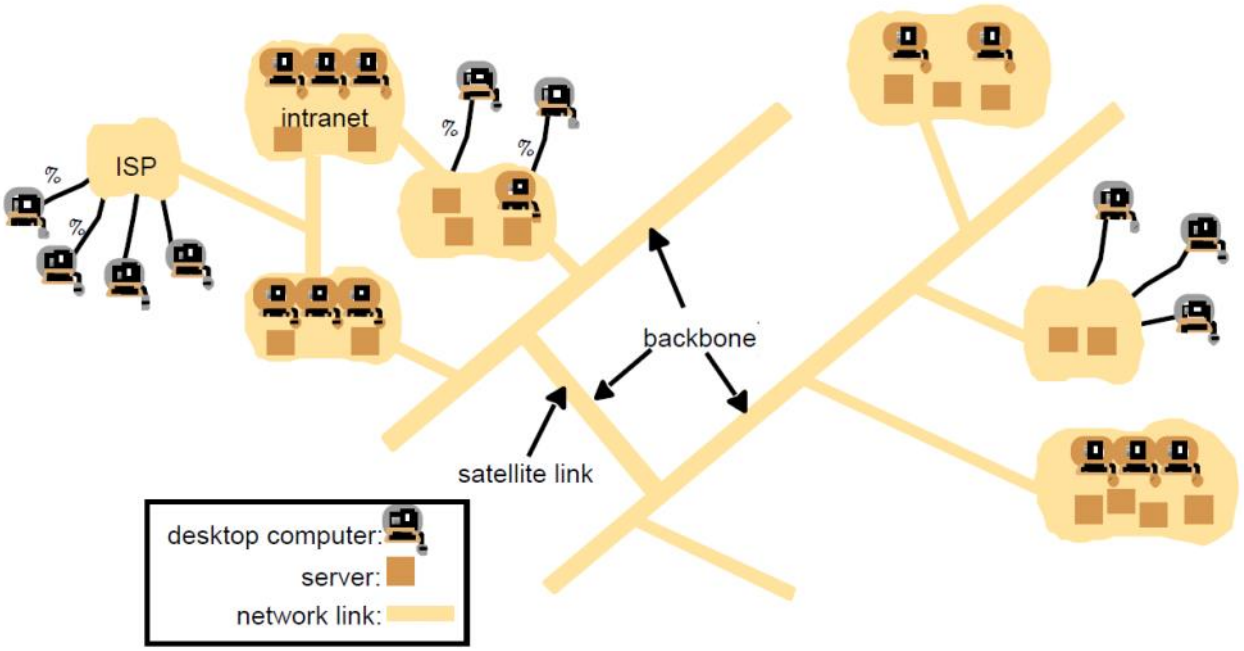
1. Key textbook and related books.
2. Course information.

Chapter1. Characterization of Distributed Systems

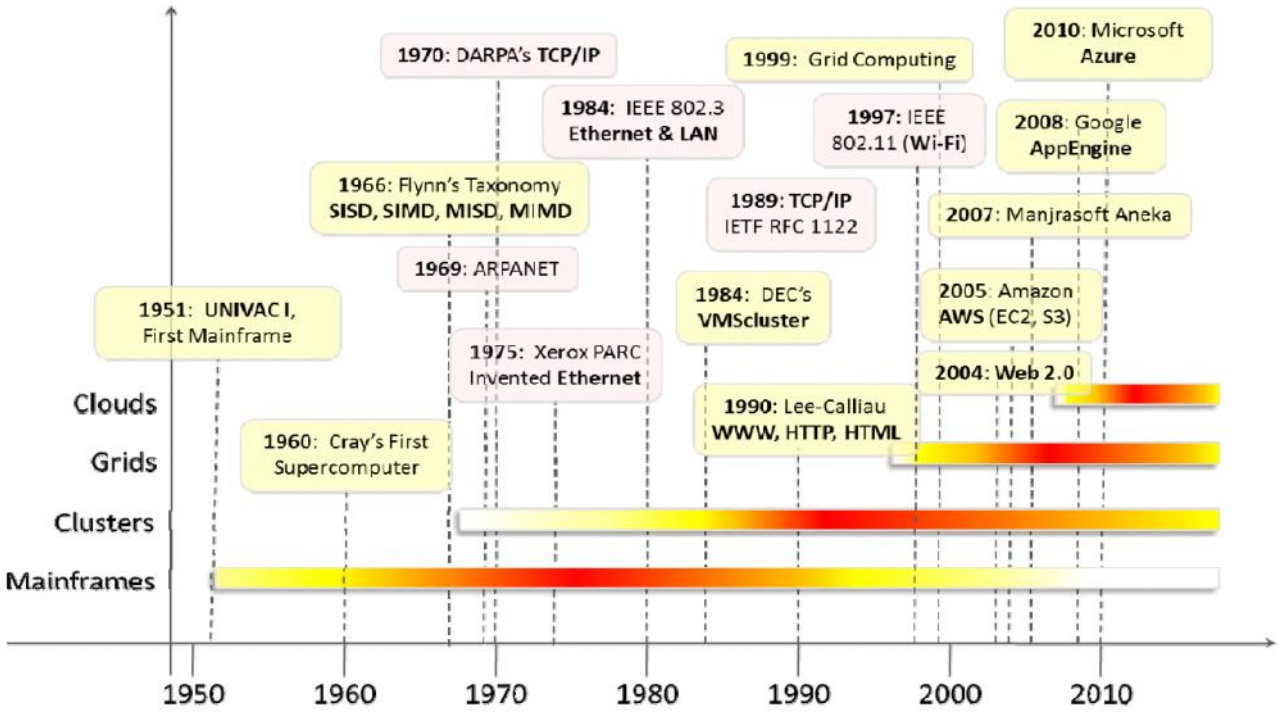
2019年2月20日 8:19

Note Taking Area

A typical portion of the Internet



- Different kinds of servers: email, print, router/firewall, file, web, and other.
- the annals of distributed system:



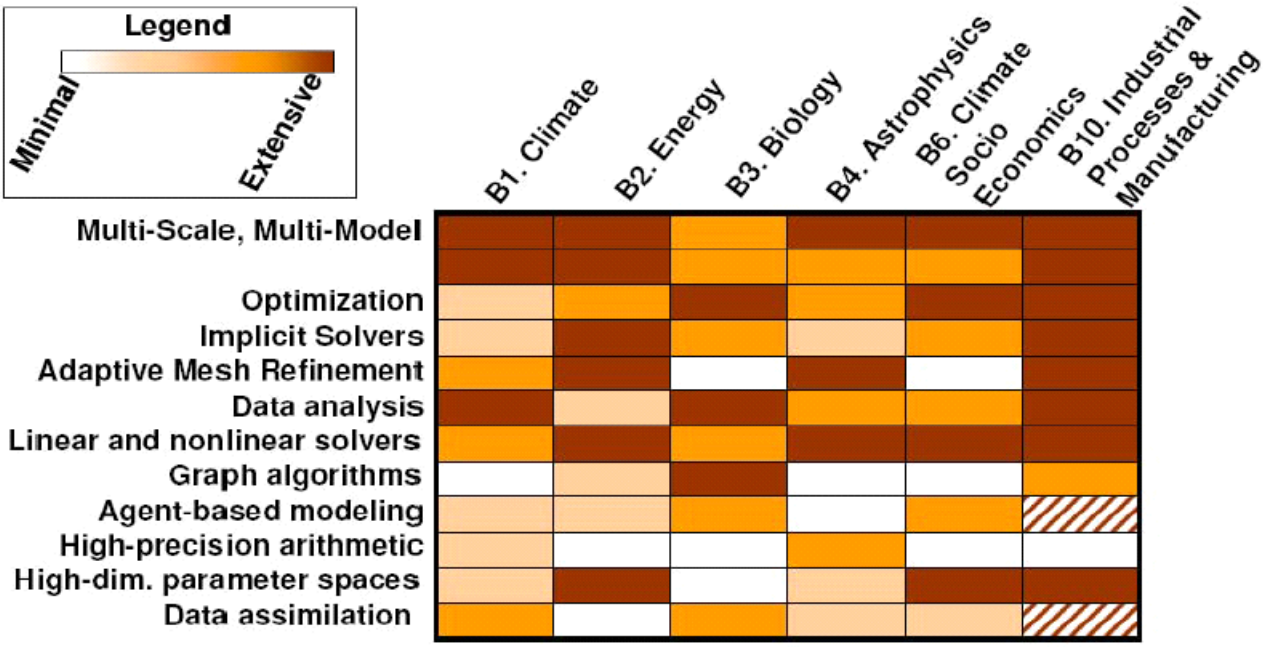
  - Map-reduce: remove invalid data.
  - SISD, SIMD, MISD, MIMD: Single/multi instruction flow, and single/multi data flow.
    - 单/多指令流、单/多数据流。

Hardware introduction

- The GPU isn't like CPU, which is better at float computing.

- The cost of CPU to access different region in CPU is different.

Trend in computing



- Many areas uses cloud machine, but white blocks mean there doesn't need cloud machine.
- The limit factor of super computer is power.

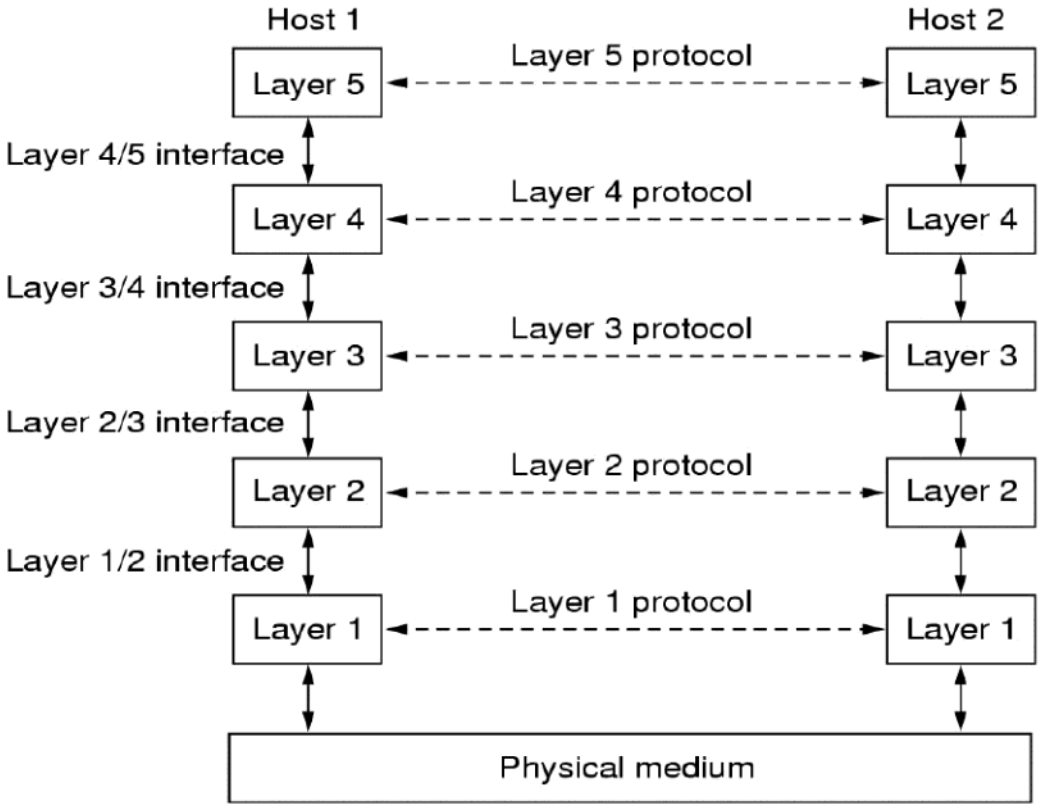
Grid standards and middleware

- HPC Systems: high-performance computing.
  - Homogeneous nodes -> high speed -> clusters or MPPs(centralized control) -> disparate clusters -> computational and data grids -> web2.0 service / internet clouds(virtualization) / internet of things(RFID and sensors)
- HTC Systems: high-throughput computing.
  - Disparate nodes -> P2P network(distributed control) -> geographically sparse -> computational and data grids -> web2.0 service / internet clouds(virtualization) / internet of things(RFID and sensors)
- P2P: peer to peer.
- MPP: massively parallel processors.

Distributed systems

- Definition: a system in which components located in networked computers communicate and coordinate their actions only by passing messages.
  - Motivation: sharing of resources (example web servers and web browsers).
- Computer network: a collection of interconnected autonomous computers.
  - Generality: built from general purpose hardware, instead of any particular application.
- Relationship between computer network and distributed system:
  - Transparency, and DS is a software runs on the top of CN.

CN layer architecture



- Purpose: offer a communication services to higher level layers.
- Two interfaces in each layer:
  - Peer-to-peer: defines the form and types of messages exchanges between peers.
  - Services: defines the primitives (operations) that a layer provides to the layer above it.
  - Layering is non-linear.

Internet architecture

- ISO OSI seven layers:
  a. Physical: transmission of raw bits onto the communications medium.
  b. Data link: reliable transmission of frames, flow control, arbitration.
  c. Network: packet switching, routing congestion control.
  d. Transport: process-to-process channel, node-to-node connection, provides user services, flow control, multiplexing.
  e. Session: Protocols necessary to establish and maintain a connection or session between 2 end-users, orderly communication (dialog management, synchronization points, activities).
  f. Presentation: communication of information rather than data, code and number conversion, transmission of sophisticated data structure, and data compression.
  g. Application: electronic mail, ftp, telnet, distributed system, and client-server.
- TCP/IP four layers:
  a. Host-to-network layer: OSI physical and data link layers.
  b. Internet layer: OSI network layer - Internet Protocol / IP.
  c. Transport layer: transmission control protocol / TCP & user datagram protocol / UDP.
  d. Application layer.

Cue Column

Instruction flow and data flow

- SISD: old serial computer, and in one clock period, one data flow is operated.
  - Von Neumann machine, IBM PC, and 8 bit computer.
- SIMD: one instruction flow to operate multi data flow.
  - Digital signal processing, photo processing, and multimedia information processing.
- MISD: just a concept and never show up.
- MIMD: carry out multi instruction flow and process multi data flow at same time.
  - Multi cores computer.

Summaries

1. A typical portion of Internet.
2. Hardware introduction.
3. Trend in computing.
4. Grid standards and middleware.
5. Distributed systems.
6. CN layer architecture.
7. ISO OSI architecture and TCP/IP architecture.

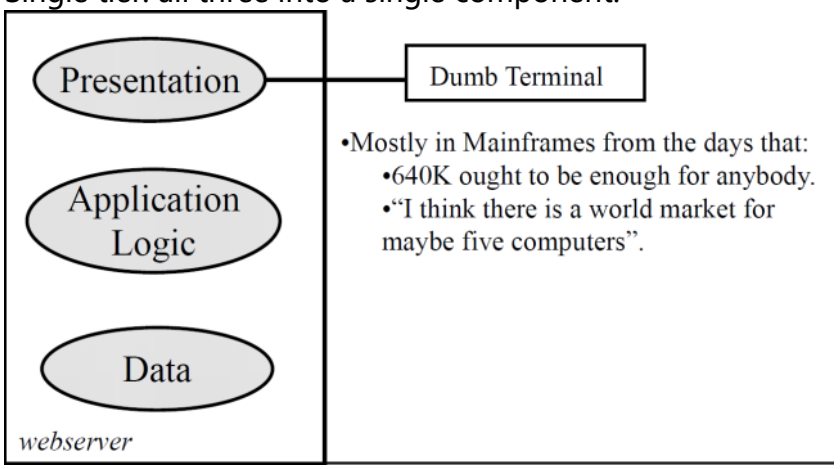
Chapter2. System Models

2019年3月5日 11:24

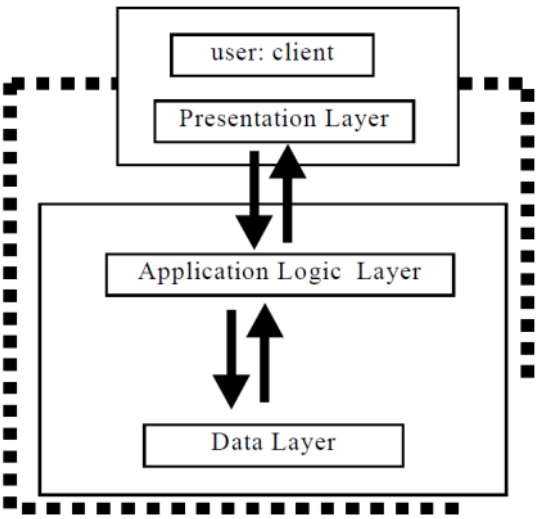
NOTE TAKING AREA

e-Business application

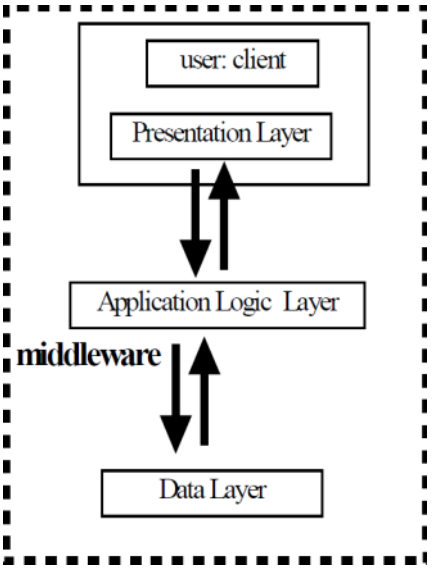
- Architecture: structure in terms of separate specified component.
  - User: client <-> presentation layer <-> application logic layer <-> data layer.
- Presentation layer:
  - Interaction with user.
  - User submits operations and get responses.
  - The boundary of presentation layer and client can be thin, like Java Applet.
- Application logic layer (business process, business logic):
  - Process data before delivery of the result (implementation).
  - All those programs and module involved in processing the operation.
  - Single tier: all three into a single component.



  - 2-tier architecture: birth of PC.



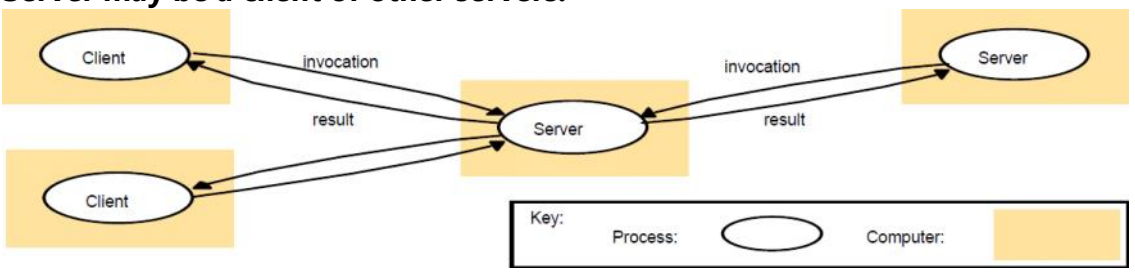
    - Remote procedure call, fat client and thin client.
  - Three tier: infrastructure that supports the development of ALL is called a middleware.
    - ALL: application logic layer.



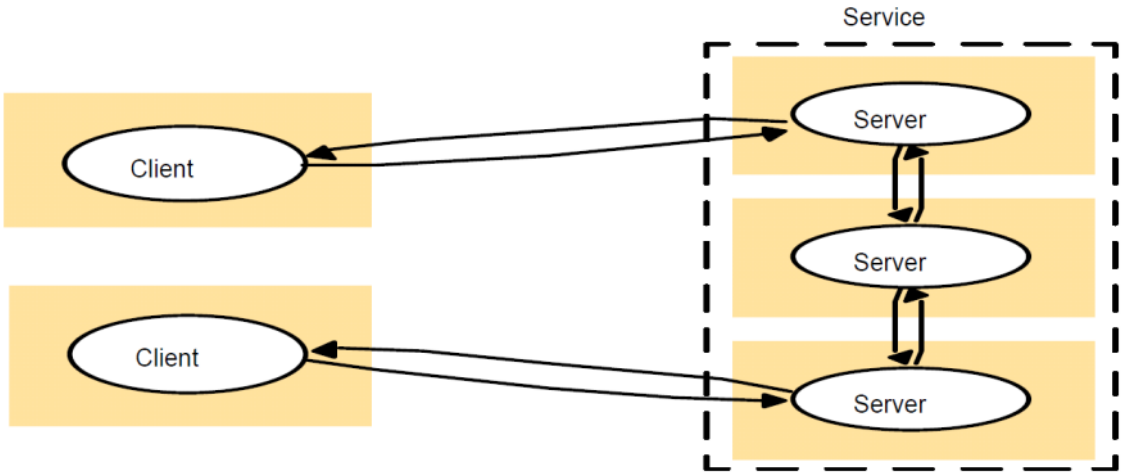
    - Data layer resulted in better interfaces.
  - N tier: many distributed systems (3-tier) interacting.
- Service layers in a distributed system
  - Hardware + network + OS = platform.
  - Middleware:
    - Mask heterogeneity.
    - Programming model.
    - Provides building blocks.

Architectural models

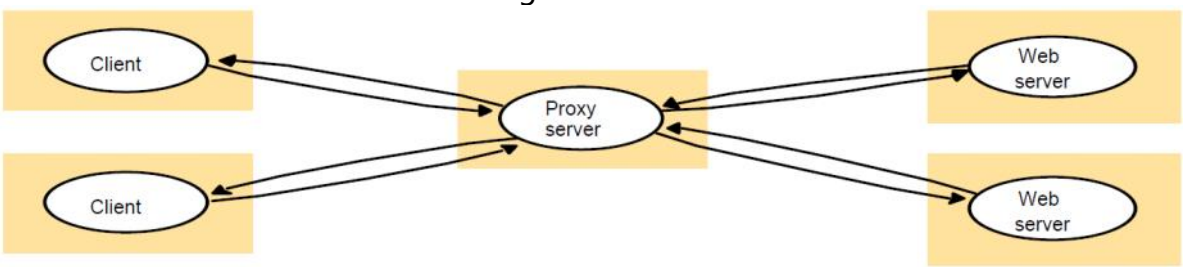
- Architectural models of distributed systems
  - Functions of components.
  - The placement of components across a network.
  - Inter-relationships between components.
- The client-server model:
  - Server: *process* that accepts to perform a service and *responds* accordingly.
  - Client: invokes services (*remote invocation*).
  - Client **object** invoke a method upon a server object.
  - **Server may be a client of other servers.**



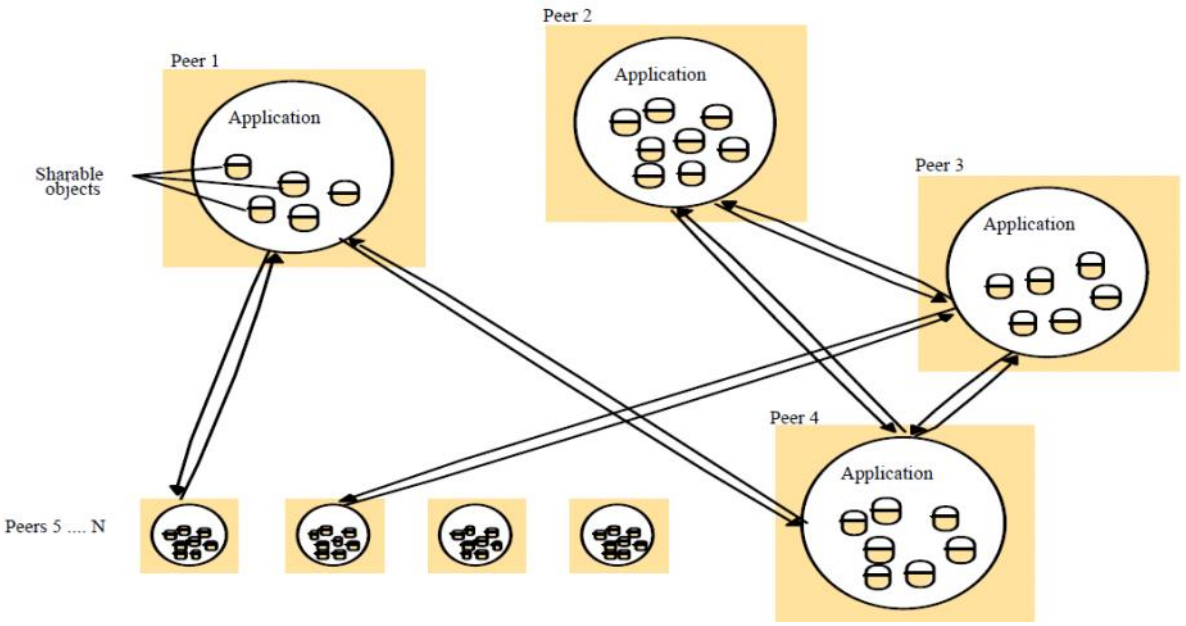
- Multiple servers:
  - Partition services (web servers).
  - Replication for performance and fault tolerant.



- Proxy servers and caches:
  - Cache: a store of recently used data objects that is closer than the objects themselves.
  - Proxy server: a shared cache of web resources for the client machine at a site or across sites.
    - Access remote web servers through a firewall.



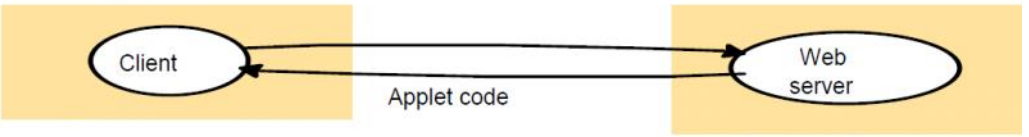
- Peer processes:
  - All processors play similar roles, and cooperate as peers to perform a distributed activity.
  - Reduce server bottleneck, but has consistency and synchronization issues.



Variations of the client-server model

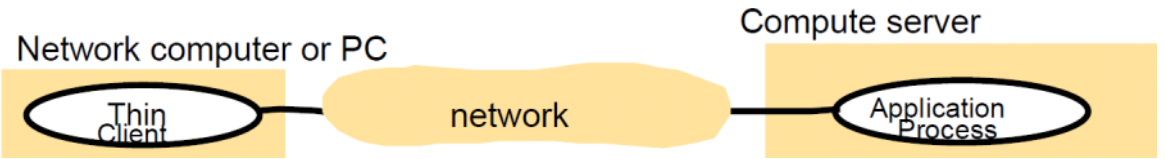
- Variations: use mobile code, for low cost computers with limited hardware resources, and need to add or remove mobile devices.
- Applets: mobile code example:

a) client request results in the downloading of applet code



b) client interacts with the applet

- Thin clients: limited hardware resources:

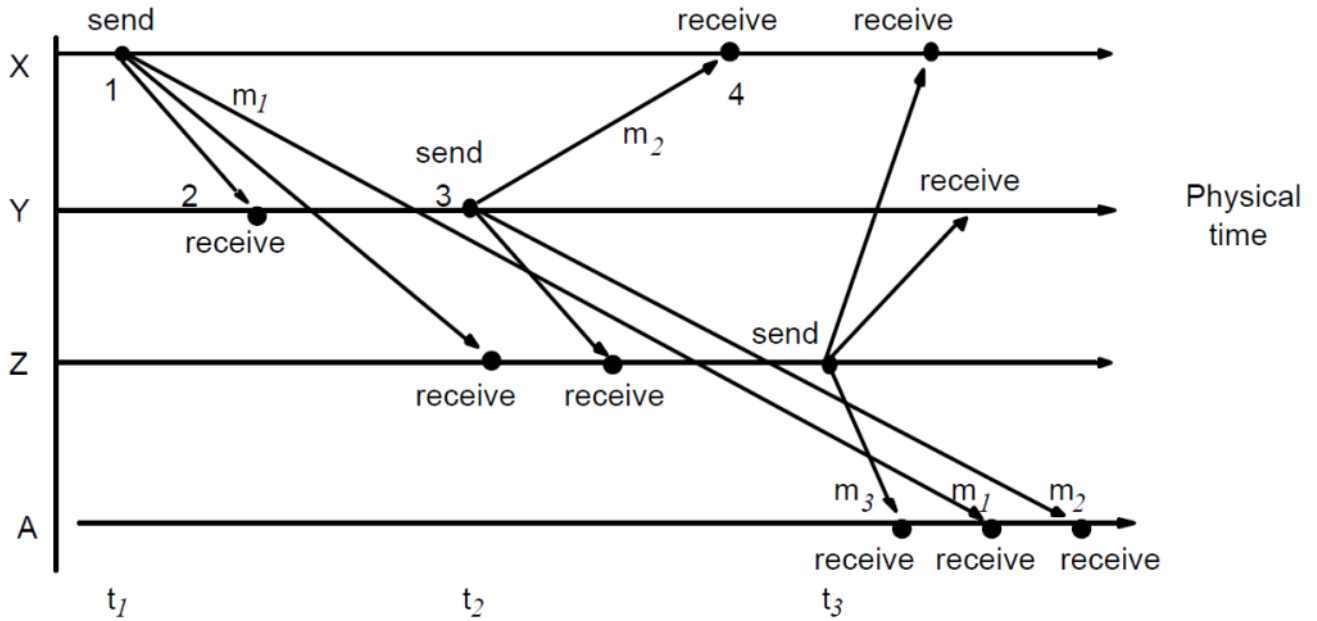


Design requirements for DSs

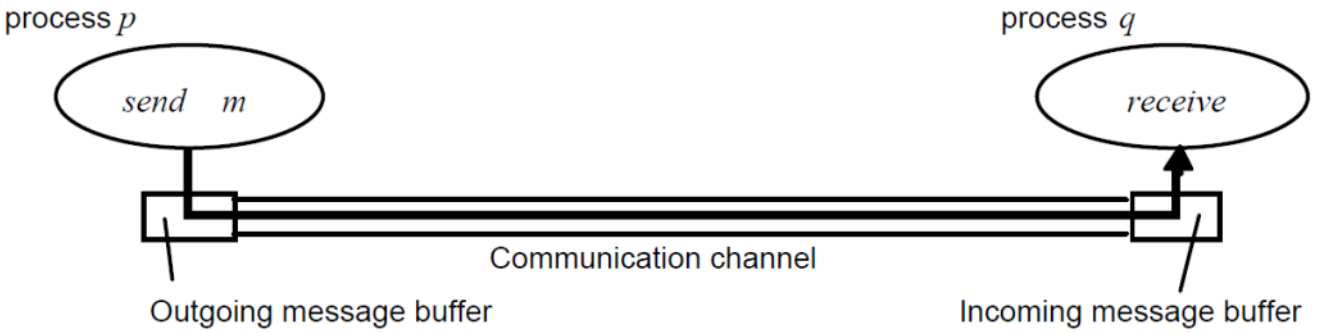
- Judging how good the architecture is: performance, quality of service, dependability.
- Performance: responsiveness, throughput, load balancing.
- Quality of Service (QoS):
  - Non-functional properties experienced by users:
    - Deadline properties: hard / soft deadlines.
  - Adaptability: adapt to changing system configuration.
- Dependability: correctness, fault-tolerance, security.

Fundamental models

- Questions:
  - What's the main entity in the system?
  - How do they interact?
  - What are the characteristics that affect their individual and collective behavior?
- Purpose: specify assumptions, make generalizations.
- Interaction model: distributed algorithms - including communication, important factors, synchronous.
- Event ordering: considering a mail system:



- Failures: omission, arbitrary, and timing failures.



- Omission and arbitrary failures:

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

- Timing failures:

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

CUE COLUMN

SUMMARIES

1. An example: e-Business application.
2. Architectural models.
3. Variations of the client-server model.
4. Design requirements for DSs.
5. Fundamental models.

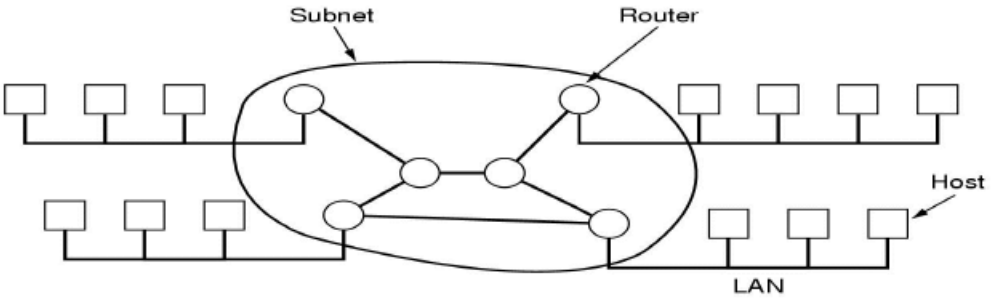
Chapter3. Interprocess Communication

2019年3月13日 8:14

NOTE TAKING AREA

Network architecture

- Definition of network architecture: framework for designing and implementing networks.
- Components: **software** - protocols and services, **hardware** - transmission technology, media and devices, scale of LANs, MANs, and WANs, topology.
- Subnet: a part of the whole network,

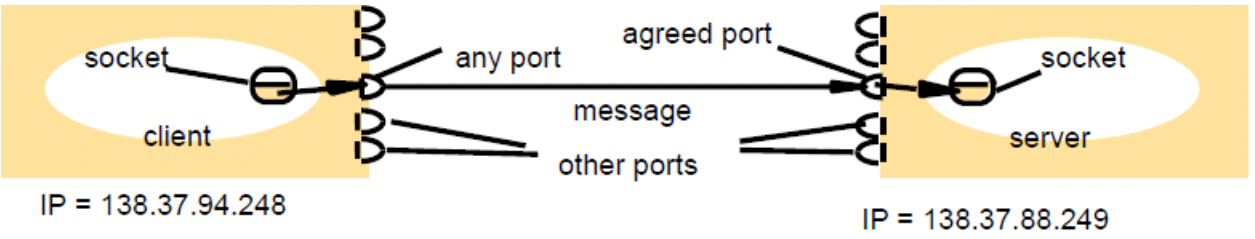


Layered architecture:

- *The purpose of each layer is to **offer a communication services to higher level layers**.*
- Interfaces in each layer:
  - **peer-to-peer interface** - defines the form and types of messages exchanged between peers (indirect communication).
  - **service interface** (defines the primitives (operations) that a layer provides to the layer above it.
  - Layering is non-linear.
- Protocols: the functionality encapsulated within each layer.
  - Refers to **interfaces** and the **objects that implement those interfaces**.
  - Protocol and service: service is the set of **primitives** provided to the higher layer, while protocol **implementation these primitives**.
- Internet architecture includes ISO OSI architecture, TCP/IP architecture. *I'll escape this part that you can find the knowledge here from computer network lesson.*

Interprocess communication

- Message passing model: send - receive.
  - Synchronous communications: block and asynchronous communications: blocking (receive) and non-blocking (send, receive).
- Reliability: validity (guarantee messages delivered and doesn't lost), integrity (message arrives uncorrupted and without duplication).
- Sockets and ports:
  - *Sockets provide an abstract endpoint for inter-process communication (UDP, TCP).*
  - Destination: internet address (names rather than numbers), port (within a computer).
  - Socket must be bound to a **local port** and an **internet address**.
  - A process may **use multiple ports** but **cannot share ports**.

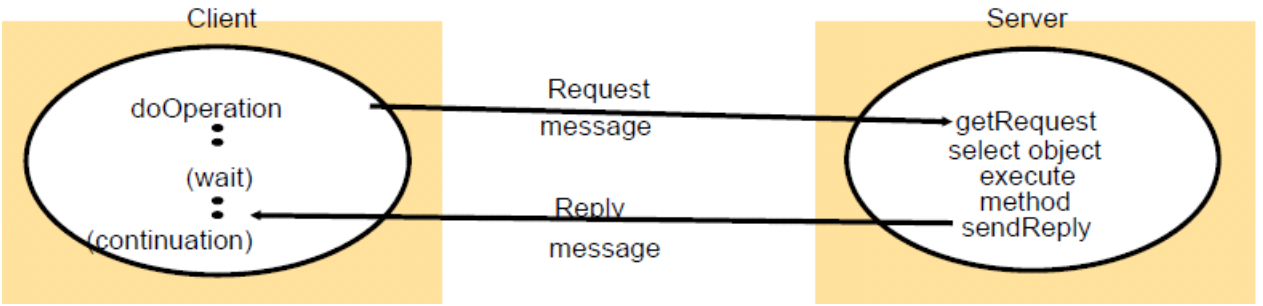


Remote object reference

- A valid identifier for an object throughout the distributed system should be **unique**, and be **passed as argument**, with **external representation**.

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

- Request-reply protocol



  - Issues of this protocol:
    - Message identifiers (sequence and sender id).
    - Failure model such as timeouts, lost reply messages.
    - Duplicate request messages.
    - History for retransmissions (log).
- UDP sockets: receive method returns the IP address + port of sender so that a reply can be sent.
  - Message size: specifies a byte-array to receive the message.
  - Non-blocking send & blocking receive (timeouts and threads for deadlocks)
- TCP sockets: *provides an abstraction of a stream of bytes to which data write and read. It hides the following:*
  - Message sizes, lost messages, flow control, message duplication and ordering, message destinations.
  - Connection: client-server model.
    - Client: creates a stream socket bound to a port and asks for a connection to a server port.
    - Server: creates a listening socket bound a port and waits to accept connect requests.
  - Each socket is **both for input and output**.

Java API for network transmission

- Java API for Internet address:
  - $Class\ InetAddress$: uses DNS (Domain Name System).
  - $InetAddress\ aComputer = InetAddress.getByName("gromit.bham.ac.uk")$
  - $throws\ UnknownHostException$
- Java API for UDP sockets:
  - $Class\ DatagramPacket$: $getData, getPort, getAddress$: for receiving messages.
  - $Class\ DatagramSocket$: $send, receive, setSoTimeout, connect, SocketException$: port in parameter can be specified or free. Use for sending message, and connect to another socket.

- Java API for Datagrams
  - $Class\ DatagramPacket$: $getData, getPort, getAddress$: send / receive messages, while the messages possibly lost / out of order.

message (=array of bytes)	message length	Internet addr	port no

- Java API for TCP socket:
  - $Class\ ServerSocket$: $accept$: server port socket, and listening for connect requests.
  - $Class\ Socket$: $getInputStream, getOutputStream, UnknownHostException, IOExcep$: client port socket and connects to remote server.

Group communication

- Multicast: sends a single message **from one process to each of the members of a group of processes**.
  - Fault tolerance based on **replicated services**.
  - Finding discovery servers in spontaneous networking.
  - Propagation of event notifications.
- IP multicast: multicast group is specified by a class D Internet address.
  - Membership is **dynamic**, to join make a socket.
  - Programs using multicast use **UDP** and send datagrams to multicast addresses and (ordinary) port.

CUE COLUMN

TCP Java API example

- TCP client connection, sending request and receiving reply:

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
        }
}
```

- TCP server make connections and echoes requests:

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
            try{
                        int serverPort = 7896;
                        ServerSocket listenSocket = new ServerSocket(serverPort);
                        while(true) {
                                    Socket clientSocket = listenSocket.accept();
                                    Connection c = new Connection(clientSocket);
                        }
            } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
```

- Continued:

```
class Connection extends Thread {
            DataInputStream in;
            DataOutputStream out;
            Socket clientSocket;
            public Connection (Socket aClientSocket) {
               try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out =new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
            }
            public void run(){
               try {                                          // an echo server
                        String data = in.readUTF();
                        out.writeUTF(data);
               } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
               } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
               } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
            }
}
```

SUMMARIES

1. Network architecture.
2. Layered architecture, protocols.
3. Interprocess communication, sockets.
4. Remote object reference.
5. Java API for network transmission.
6. Group communication.

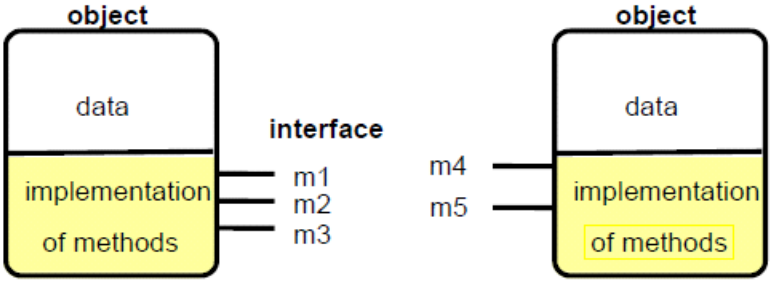
Chapter4. Distributed Objects and Remote Invocation

2019年3月20日 8:06

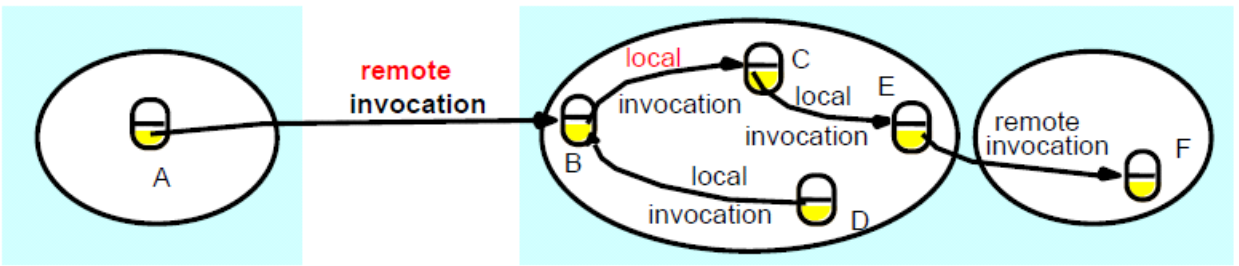
NOTE TAKING AREA

Distributed applications programming

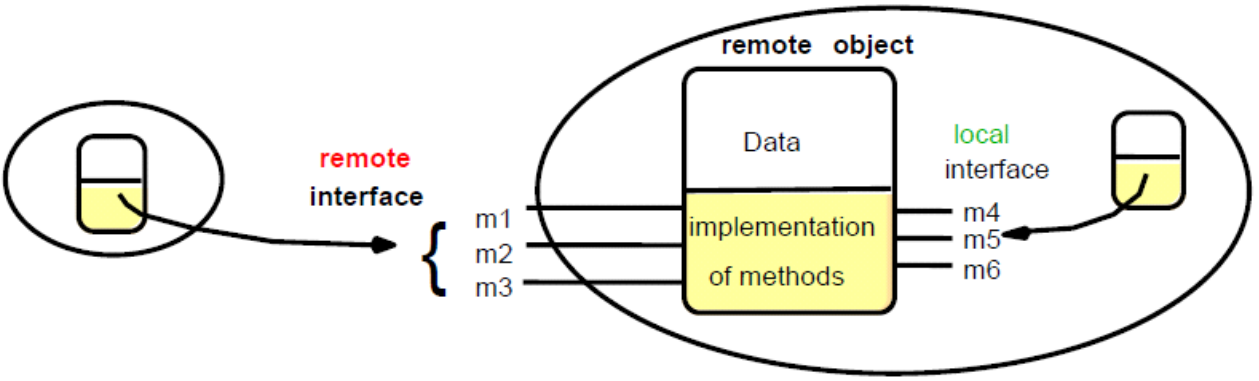
- Distributed objects model:
  - Middle layers: **RMI**, **RPC**, and **events**. As well as **request reply protocol**, and **external data representation**.
  - Objects: data (attributes) + operations (method).



    - State of objects: value of its attributes. (Encapsulating)
    - Interact via interfaces: define types of arguments and exceptions of methods.
- The object (local) model:
  - OO programs: a collection of interacting objects.
  - Distributed object model:



    - Objects extend with remote objects and methods.
  - Remote object reference: in distributed system, must be unique in **space** and **time**, error returned if accessing a deleted object, and can allow relocation.
  - Remote interfaces: **Implements the methods** of its remote interface.
    - Interfaces specify externally accessed.
    - Parameters contain input, output, or both, instead of call by value, call by reference.
    - No pointer and constructors.

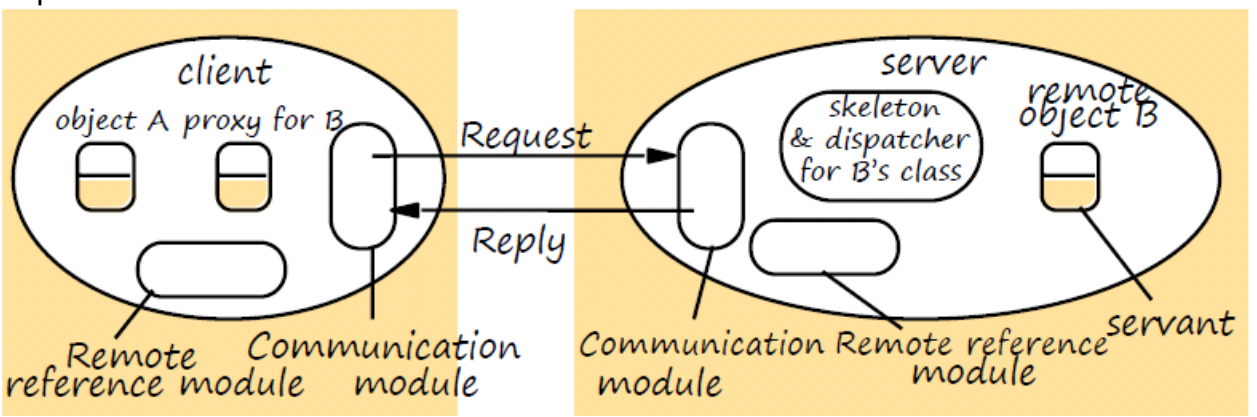


  - CORBA: interface definition language (IDL).
  - Java RMI: as other interfaces, keyword Remote.

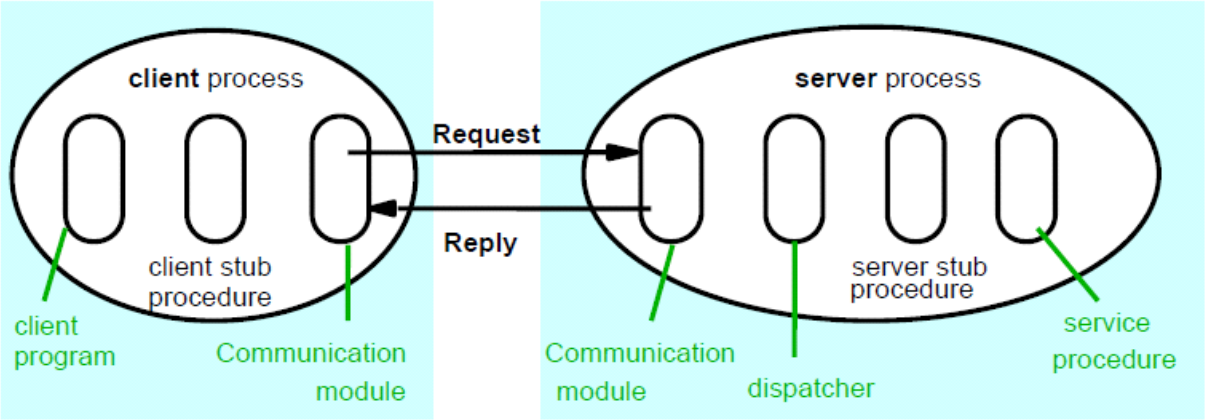
- RMI, invocation semantics:
  - Handling remote objects: **exceptions** and **garbage collection**.
  - Design issue for RMI: **invocation semantics**, maybe **invocation not guaranteed**, at least once **result or exception Sun RPC**, At most once **Java and CORBA**.

Fault tolerance measures			*Invocation 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*

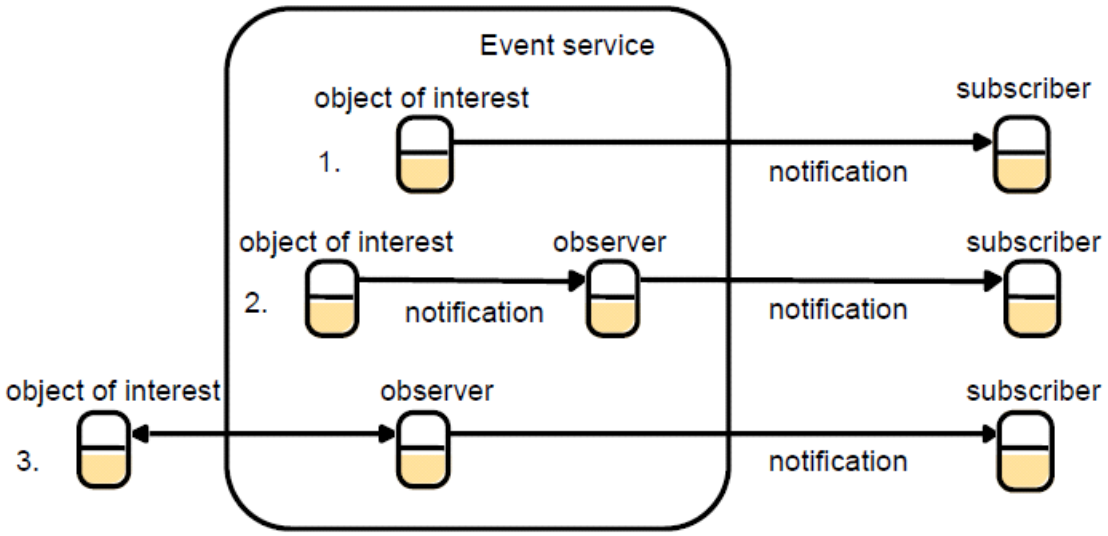
  - Implementation of RMI:



    - Application-level object A invokes a remote method in application-level object B.
    - Communication model: implements the **request-reply protocol**.
    - Proxies: transparent to clients by behaving like a local object to the invoker. It sends a message.
    - Remote reference module: creates remote object references and proxies.
      - Remote object table: entries for all **remote objects held by the process** and **entries for all local proxies**.
    - Servants: provides the body of a remote object.
    - Dispatcher: receives the request from communication module, and use the method ID to select the appropriate method in the skeleton.
    - Skeleton: Implements the methods in the remote interface, and unmarshalls the arguments in the request and invokes the corresponding method in the servant.
  - Binding and activation:
    - Binder: mapping from name to remote object references, lookup.
    - Activation: Objects active and passive, create new instance of class and initialize from stored state.
    - Activator: records location of passive and active objects, and starts server processes and activates objects within them.
  - Object location issues: persistent object stores (cf Persistent Java), object migration (use remote object reference and address), and location service (assists in locating objects, maps remote object references to probable locations).
- Remote Procedure Call (RPC):



- Events and notifications:



- Java RMI: extends the java object model to provide support for distributed objects.

Products

- Java RMI, CORBA, DCOM
- Sun RPC

CUE COLUMN

Example: Shared whiteboard

- Java Remote Interfaces *Shape* and *ShapeList*:

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;          1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

- The *Naming* class of Java RMIRegistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 1 , line 2.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 3 line 1. A remote object reference is returned.

String [] list()
This method returns an array of Strings containing the names bound in the registry.

- Java class *ShapeListServer* with *main* method:

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
                Naming.rebind("Shape List", aShapeList );                     2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

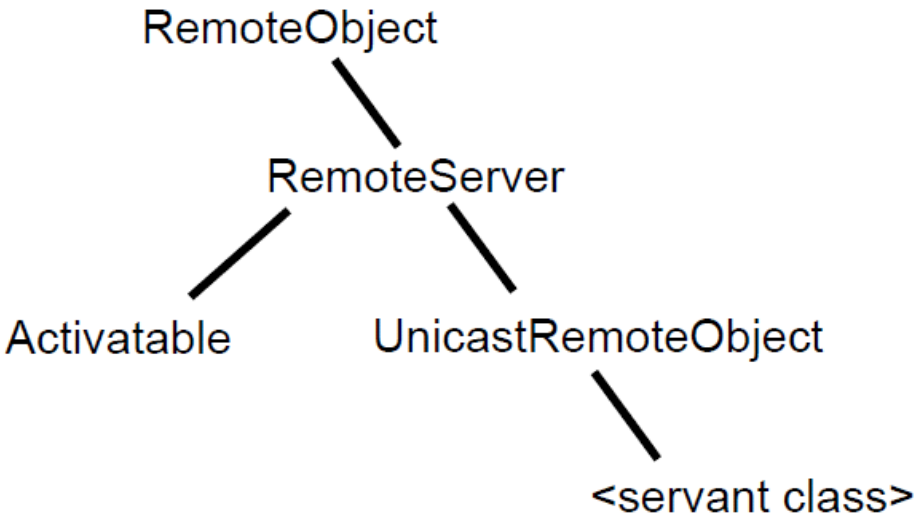
- Java class *ShapeListServant* implements interface *ShapeList*:

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;          // contains the list of Shapes          1
     private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {        2
        version++;
            Shape s = new ShapeServant( g, version);                          3
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

- Java client of *ShapeList*:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;          1
          Vector sList = aShapeList.allShapes();                                  2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

- The Classes supporting Java RMI:



SUMMARIES

1. Distributed application programming, RMI and RPC.
2. Products of remote invoke.

Chapter5. Indirect Communication

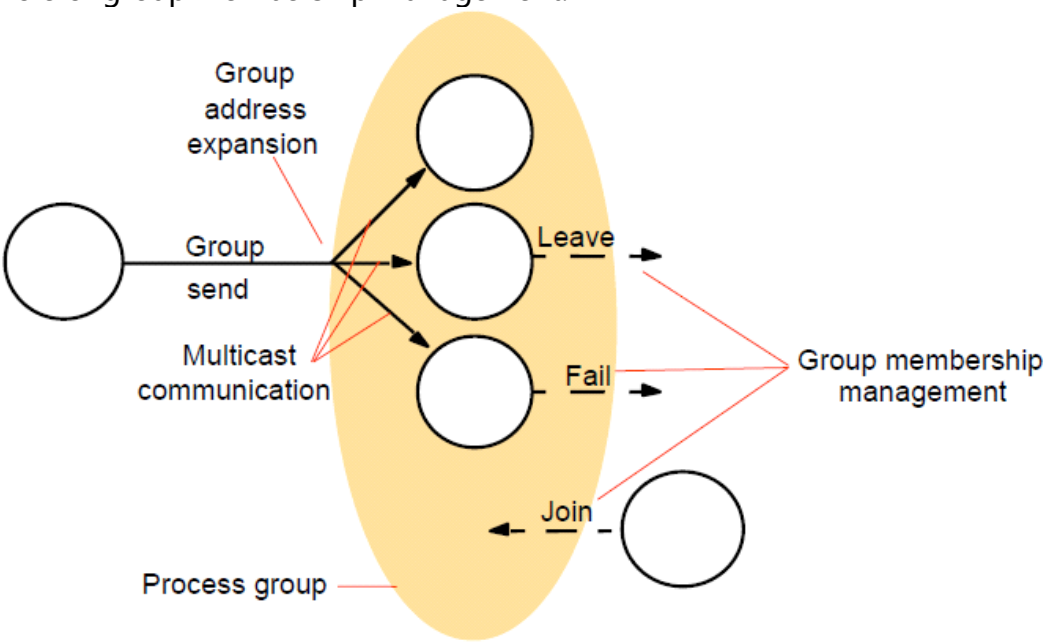
2019年3月24日 9:43

NOTE TAKING AREA

Space and time coupling

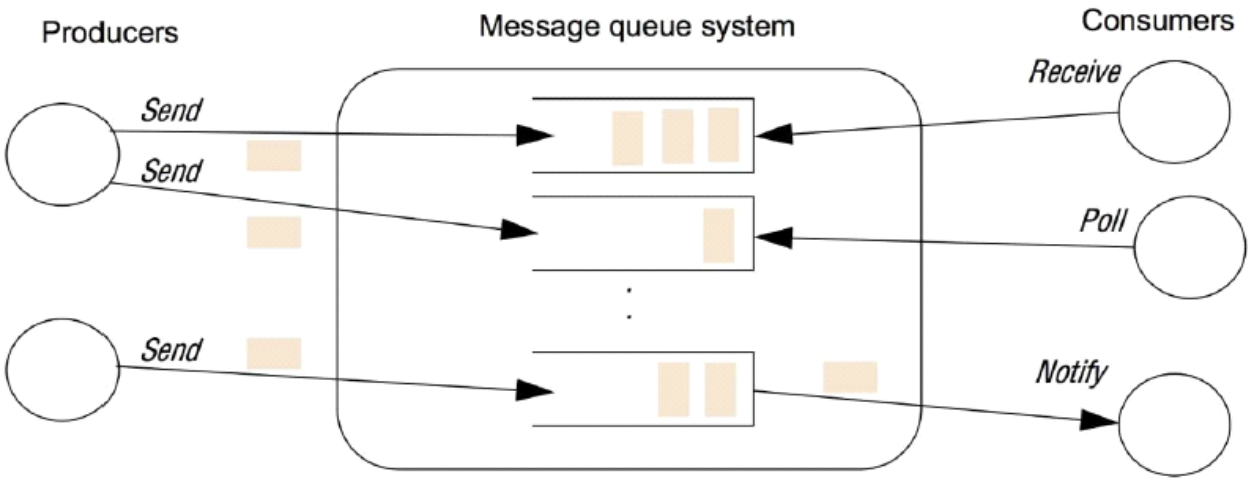
	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

- Space: whether sender knows receiver.
- Time: whether receiver exist at that moment or not.
- Open and closed groups: external sender can reach the members of group or not.
- Reliability and ordering:
  - Integrity, validity, and agreement.
  - Ordering methods: FIFO, causal, and total.
  - Group management: interface and notifying for membership changes, failure detection, and performing group address expansion.
    - Role of group membership management:



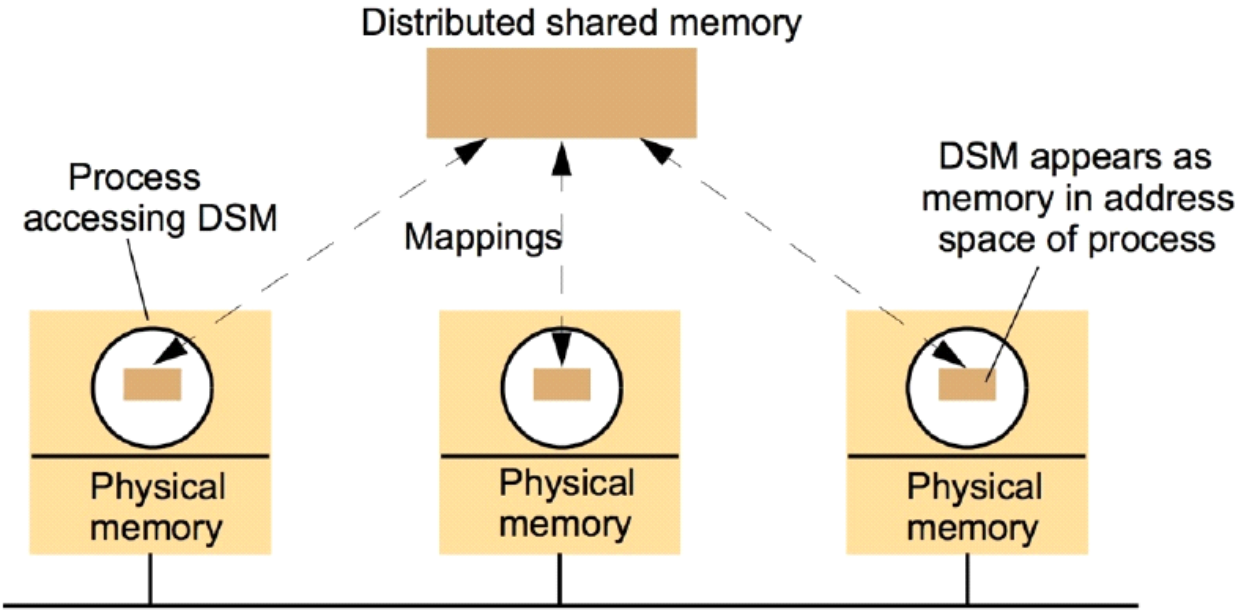
Design ideas of architecture

- The message queue paradigm:

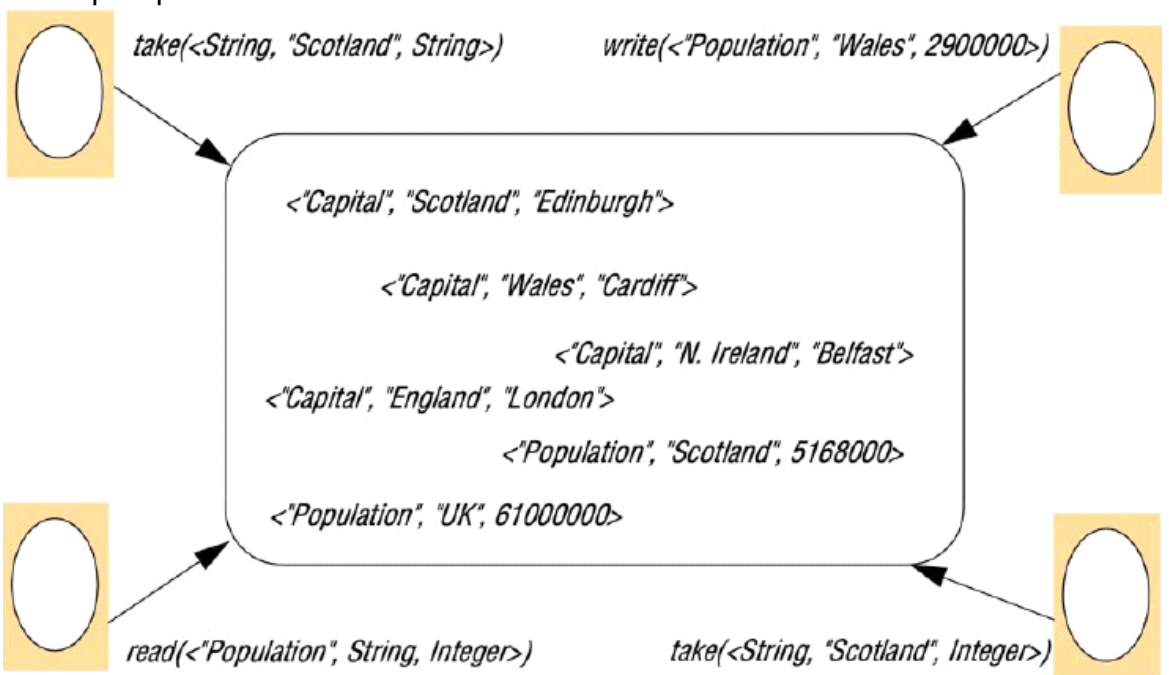


- Message queue as a mediate between producers and consumers.

- The distributed shared memory abstraction:



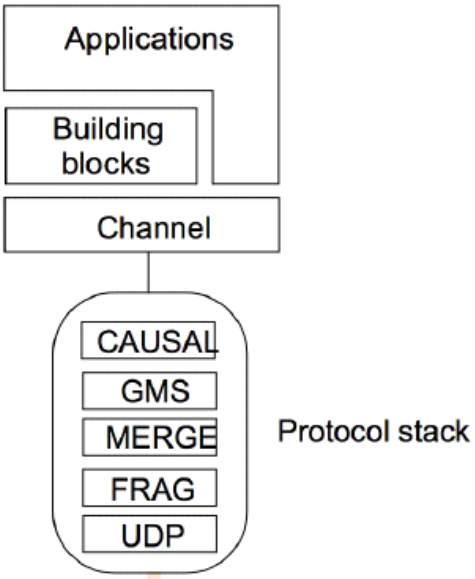
- The tuple space abstraction:



CUE COLUMN

Example: architecture of Jgroups

- The flow chart as follow:



- Class of FireAlarmJG:

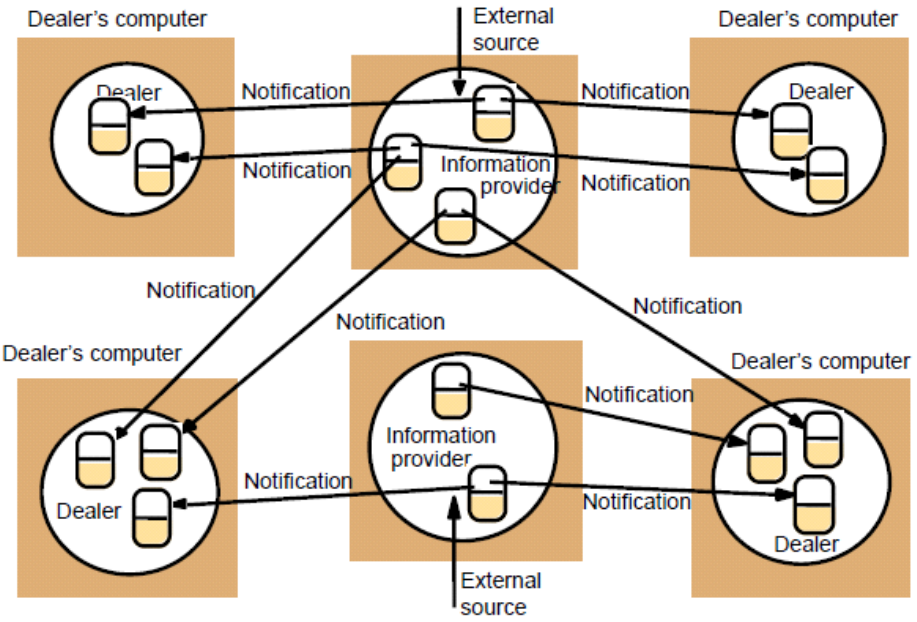
```
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() {
    try {
        JChannel channel = new JChannel();
        channel.connect("AlarmChannel");
        Message msg = new Message(null, null, "Fire!");
        channel.send(msg);
    }
    catch(Exception e) {
    }
}
```

- Class of FireAlarmConsumerJG:

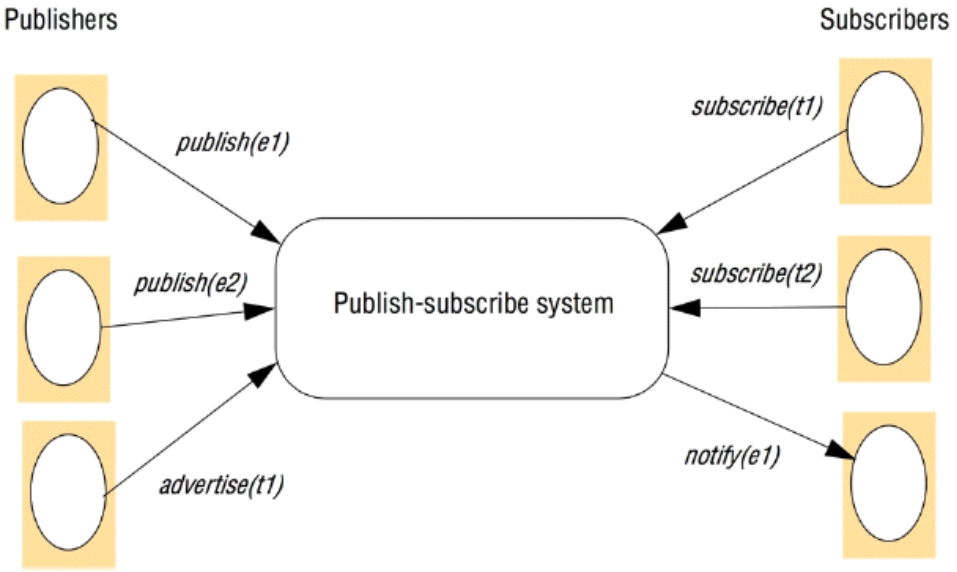
```
import org.jgroups.JChannel;

    public class FireAlarmConsumerJG {
        public String await() {
        try {
                JChannel channel = new JChannel();
                channel.connect("AlarmChannel");
                Message msg = (Message) channel.receive(0);
                return (String) msg.GetObject();
        } catch(Exception e) {
                return null;
        }
    }
}
```

Some other examples

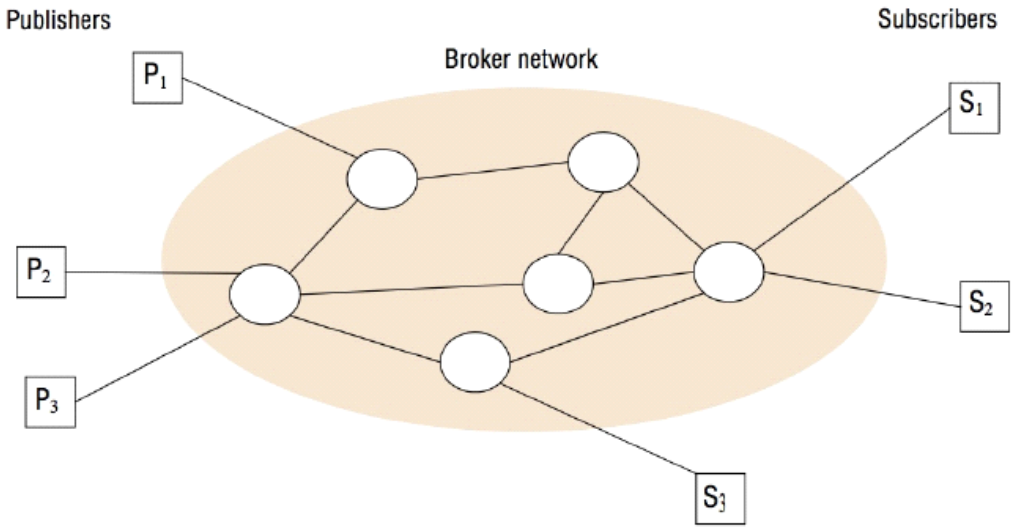
- Dealing room system:



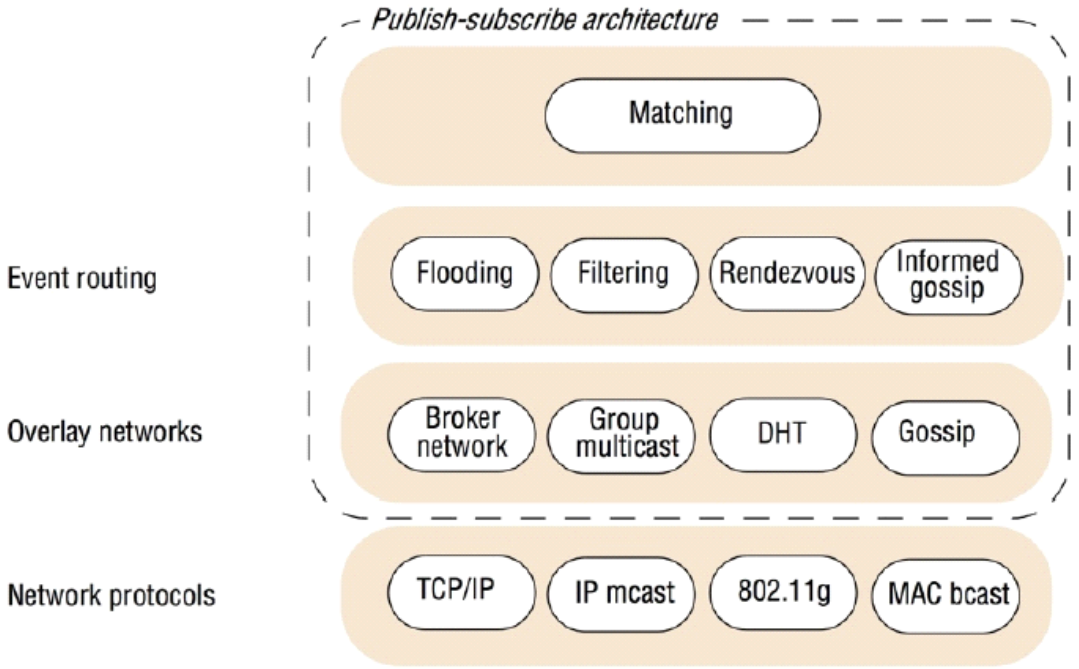
- The publish-subscribe paradigm:



  - Implementation issues:



  - Flow chart of architecture of publish-subscribe systems:



- Filtering-based routing:

```
upon receive publish(event e) from node x1
    matchlist := match(e, subscriptions)2
    send notify(e) to matchlist;      3
    fwdlist := match(e, routing);   4
    send publish(e) to fwdlist - x;  5
upon receive subscribe(subscription s) from node x6
    if x is client then    7
        add x to subscriptions;    8
    else add(x, s) to routing; 9
    send subscribe(s) to neighbours - x;10
```

- Rendezvous-based routing:

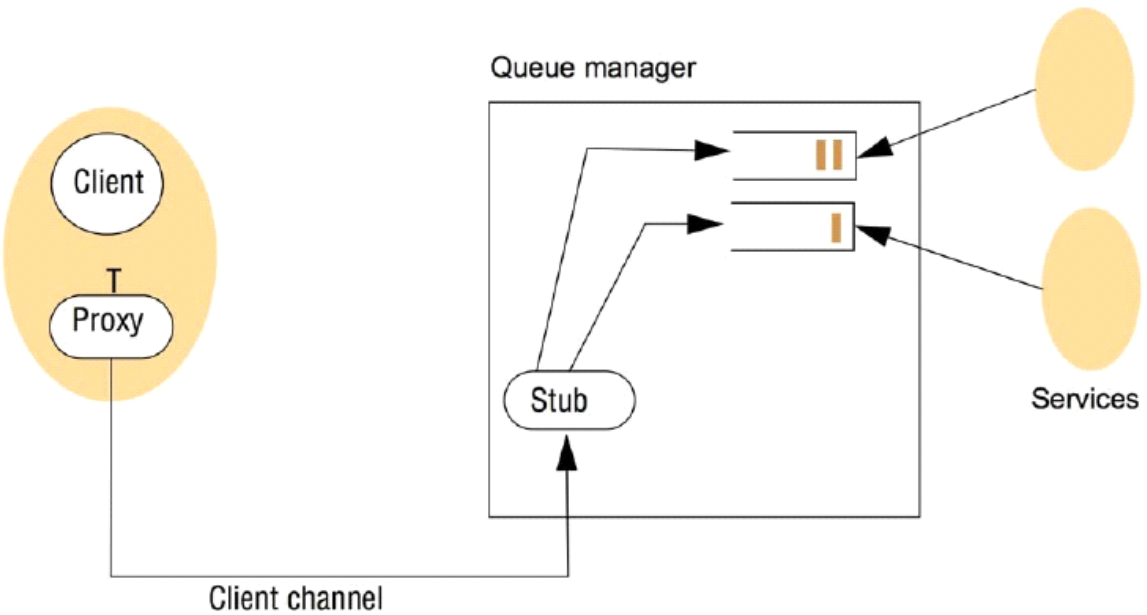
```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist :=match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

- Example publish-subscribe system:

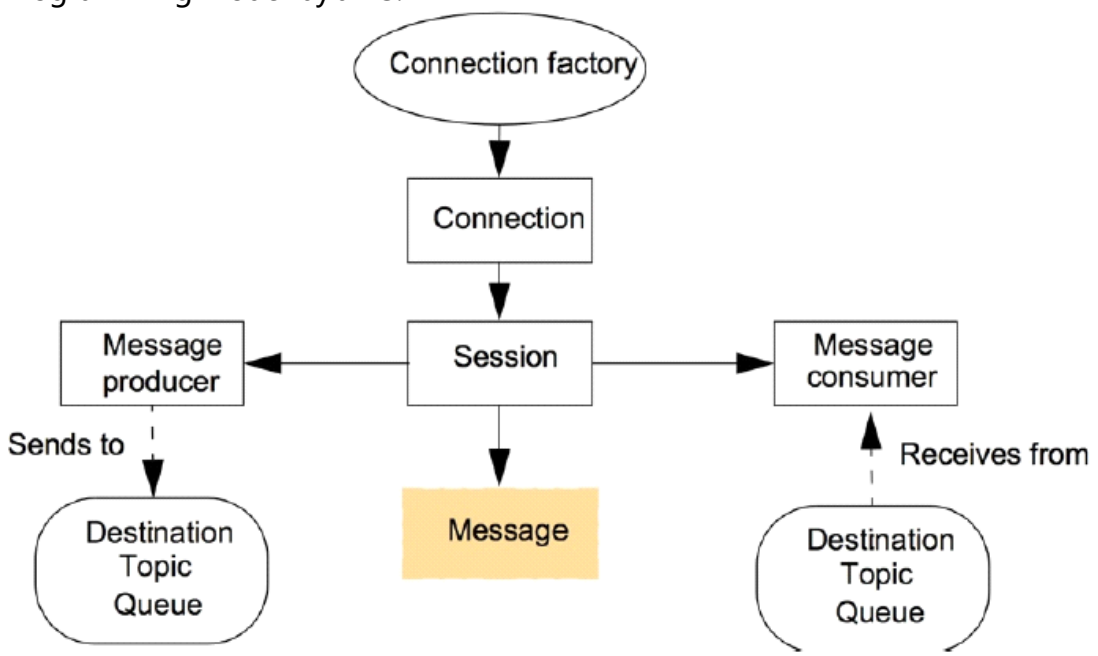
System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al.* 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al.* 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta *et al.* 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al.* 2005]	Content-based	Peer-to-peer	Informed gossip

Example of message queue

- The networked topology in WebSphere MQ:



- Programming model by JMS:



- Class of FireAlarmJMS:

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

public void raise() {
    try {                                                    1
        Context ctx = new InitialContext();          2
        TopicConnectionFactory topicFactory =     3
        (TopicConnectionFactory)ctx.lookup ("TopicConnectionFactory"); 4
        Topic topic = (Topic)ctx.lookup("Alarms");  5
        TopicConnection topicConn =     6
            topicConnectionFactory.createTopicConnection(); 7
        TopicSession topicSess = topicConn.createTopicSession(false, 8
            Session.AUTO_ACKNOWLEDGE);                                  9
        TopicPublisher topicPub = topicSess.createPublisher(topic);10;
        TextMessage msg = topicSess.createTextMessage();11
         msg.setText("Fire!");12
        topicPub.publish(message);    13
        } catch (Exception e) {     14
        } 15
}
```

- Class of FireAlarmConsumerJMS:

```
import javax.jms.*;  import javax.naming.*;
public class FireAlarmConsumerJMS
public String await() {
    try {                                                    1
        Context ctx = new InitialContext();          2
        TopicConnectionFactory topicFactory =3
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4
          Topic topic = (Topic)ctx.lookup("Alarms"); 5
          TopicConnection topicConn =         6
                topicConnectionFactory.createTopicConnection();         7
          TopicSession topicSess = topicConn.createTopicSession(false,        8
                    Session.AUTO_ACKNOWLEDGE);        9
          TopicSubscriber topicSub = topicSess.createSubscriber(topic);10
          topicSub.start();  11
          TextMessage msg = (TextMessage) topicSub.receive();12
          return msg.getText(); 13
          } catch (Exception e) {     14
                    return null; 15
   }16
}
```

API of JavaSpaces

Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace

- Class of AlarmTupleJS:

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
        public AlarmTupleJS() { }
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;}
    }
}
```

- Class of FireAlarmJS:

```
import net.jini.space.JavaSpace;
public class FireAlarmJS {
public void raise() {
    try {
        JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
        AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
        space.write(tuple, null, 60*60*1000);
     catch (Exception e) {
    }
   }
}
```

- Class of FireAlarmReceiverJS:

```
import net.jini.space.JavaSpace;
public class FireAlarmConsumerJS {
public String await() {
    try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                    Long.MAX_VALUE);
            return recvd.alarmType;
    }
        catch (Exception e) {
            return null;
        }
    }
}
```

Summary of indirect communication styles

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

SUMMARIES

1. Space and time coupling, reliability and ordering.
2. Design ideas of architecture: message queue and distributed spaces.

Chapter6. Operating System Support

2019年4月2日 10:22

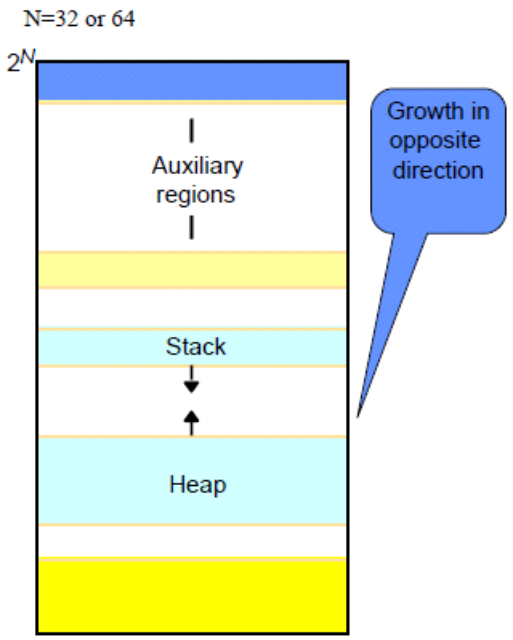
NOTE TAKING AREA

Middleware layers and system layers

- Middleware: RMI, RPC, and events. Request reply protocol, and external data representation.
  - On the bottom of applications and on the top of operating system.
- System layers: OS of kernel, libraries and servers, with **several nodes** and platform.
  - I/O management, memory management, processes, multiuser, file ownership, security, communication, and protection of processes (kernel).
- Core OS functionality:
  - Process manager: creates processes = thread + space.
  - Communication manager: sockets.
  - Thread manager: creates threads, synchronisation, and scheduling.
  - Memory manager: RAM, disk allocation.
  - Supervisor: hardware abstraction (interrupts, exceptions, caches).

Program, process, thread

- Process: a program that is currently executing, **program ≠ process**.
- Thread (lightweight process): OS abstraction of an **activity**.
- Process: **execution environment** + one or more **thread**.
  - Execution environment: threads in the same process share the same execution environment.
  - Consists of an address space, thread synchronisation mechanism, communication interface (socket), and high level resources (file and window).
- Address space: unit of virtual memory, contains one or more **regions**, text, stack, heap.

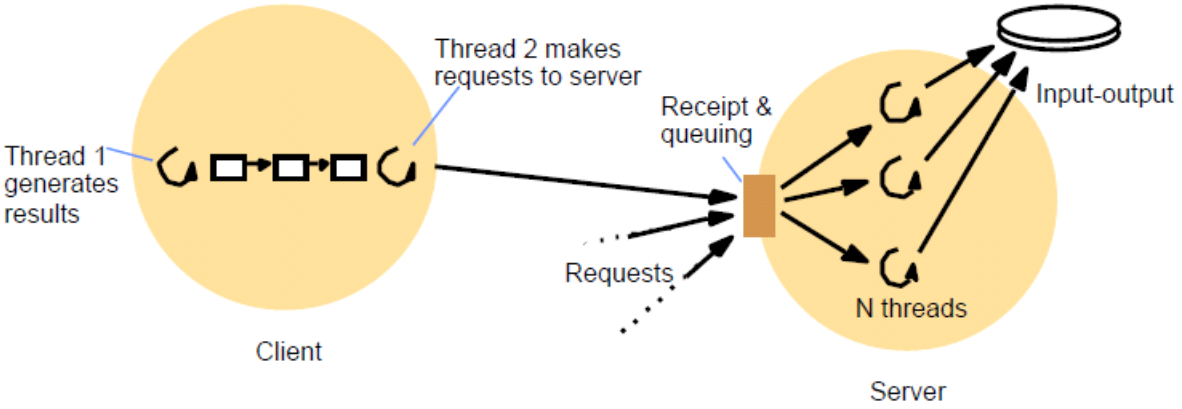


- Processes and threads:
  - Processes: historically first abstraction of single thread of activity, and can run concurrently, CPU sharing if single CPU need own execution environment.

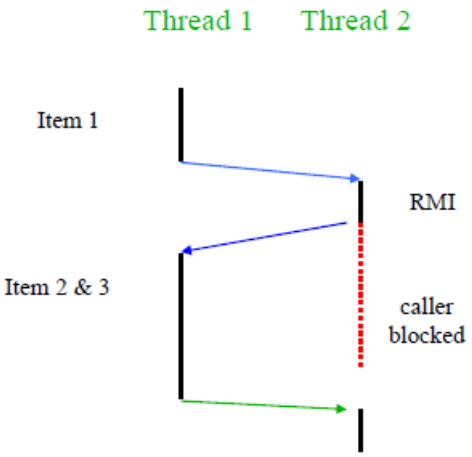
- Threads (lightweight processes): can share execution environment, and can be created / destroyed dynamically.

Threads in client server model

- Distributed system: using threads reduce waiting time, for remote invocations (blocking of invoker), for disk access (unless caching), and obtain better speed up with threads.

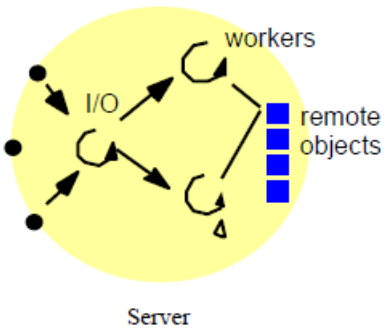


- Threads with clients: separate (data production, RMI calls to server), pass data via buffer, run concurrently, and improved speed, throughput.

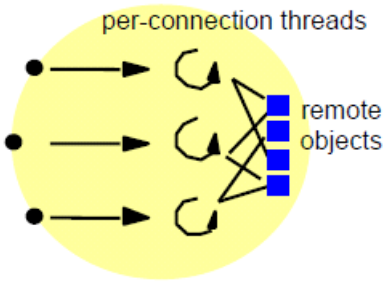


Multi-threaded server architectures

- Worker pool: fixed pool of worker threads, size does not change, can accommodate priorities but inflexible.
- Other architecture: thread-per-request, thread-per-connection, and thread-per-object.
  - Thread-per-request:

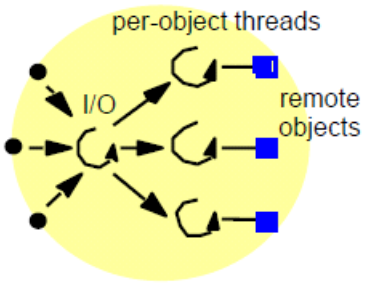


    - Spawns new worker for each request, worker destroys when finished.
    - Allows max throughput: no queuing and no I/O delays.
    - But overhead of creation and destruction high.
  - Thread-per-connection:



- Create new thread for each connection for multiple requests, destroy thread on close.
- Lower o/heads.
- But unbalanced load.

- Thread-per-object:



- As per-connection, but new thread created for each object.

- Physical parallelism: multi-processor machines (cf casper, SoCS file server).

CUE COLUMN

SUMMARIES

1. Middleware layers and system layers.
2. Program, process, thread.
3. Threads in client-server model.
4. Multi-threaded server architecture.