Communication

Slide set 2
Distributed Systems

Graduate Level

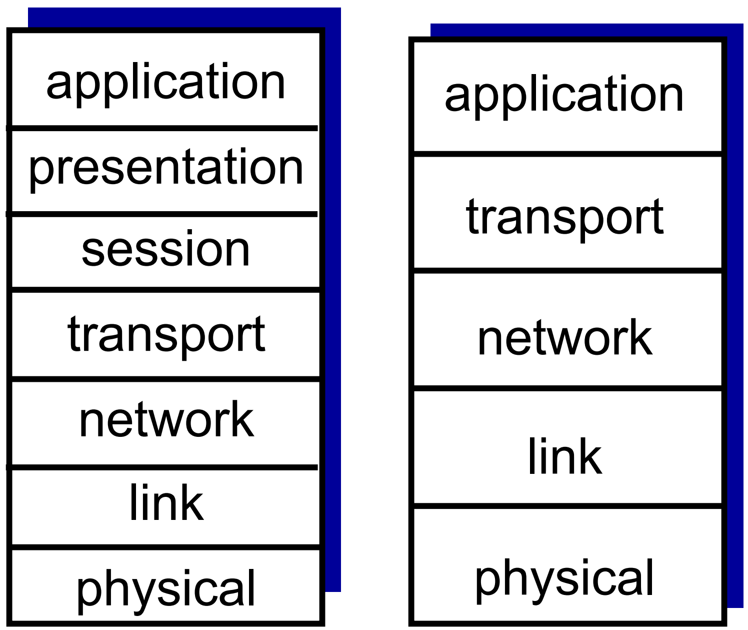
K. N. Toosi Institute of Technology
Dr. H. Khanmirza
h.khanmirza@kntu.ac.ir

Communication

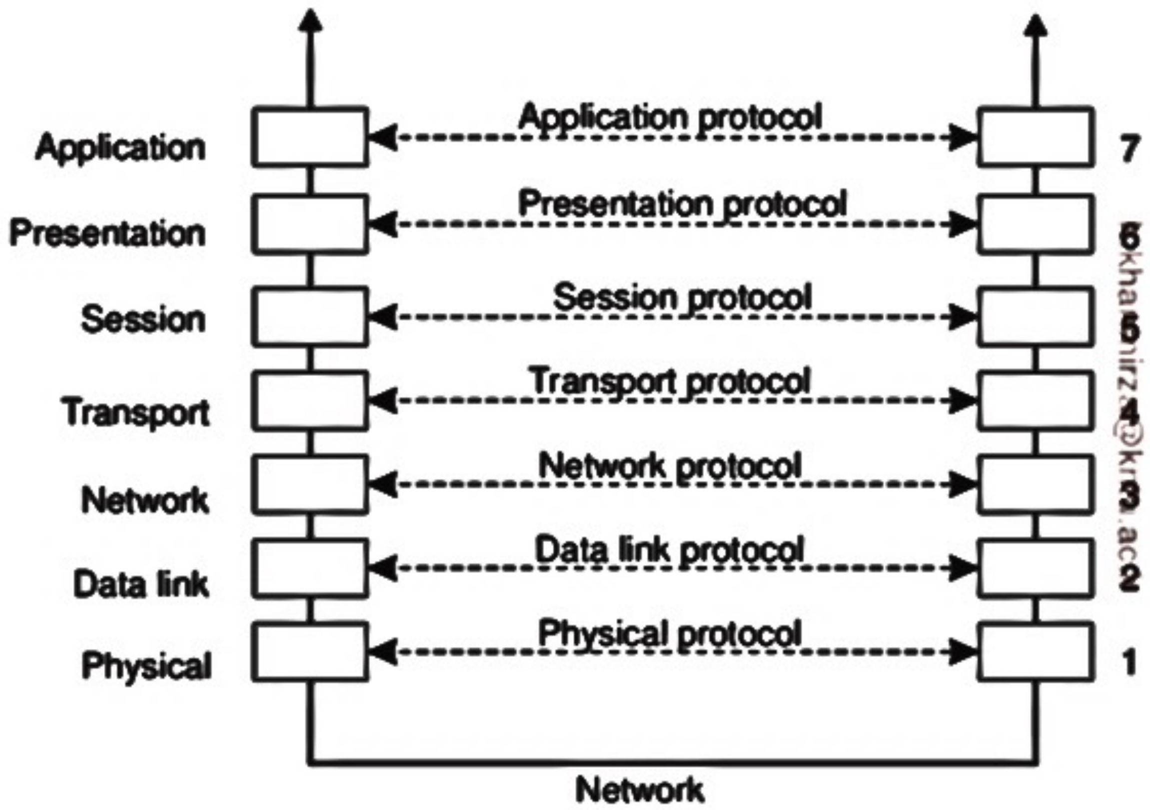
- Is the heart of all distributed systems
- End points are reside in physically different systems
- Need to create messages in a predefined formats to collaborate
  - There is no shared memory!
- Mostly rely on computer networks (socket API) to exchange messages
  - These facilities are too primitive to build complex DS systems
  - We need middleware services to perform more complex communications

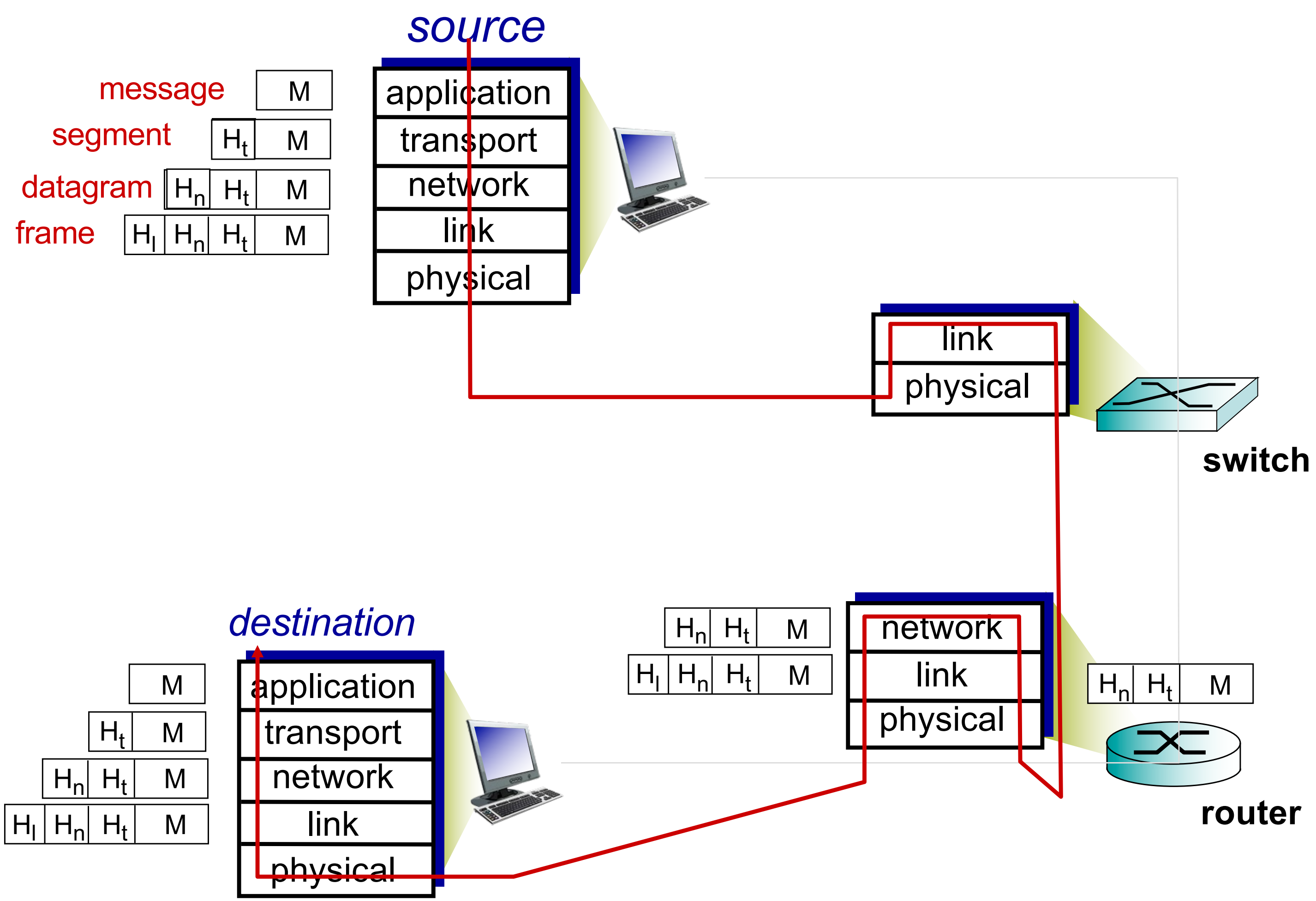
Layered Architecture

- How endpoints can talk?
  - A set of protocols that determine the meaning of the bits
- Layered Architecture
  - An approach to break a complex system into sub-systems
  - Each layer get service from underlying layer and give service to the upper layer.
- In network it is known as: Protocol Stack
  - OSI Reference Model
  - Replaced with Internet Model



Layered Architecture




source
message
segment
datagram
frame
M
Ht M
Hn Ht M
Hl Hn Ht M
application
transport
network
link
physical
link
physical
switch
destination
M
Ht M
Hn Ht M
Hl Hn Ht M
application
transport
network
link
physical
Hn Ht M
Hl Hn Ht M
network
link
physical
Hn Ht M
router

Layered Architecture

- Why Layering
  - Separation of concerns
  - Ease of developing
  - Ease of maintenance
    - Debug
    - Update
    - Test

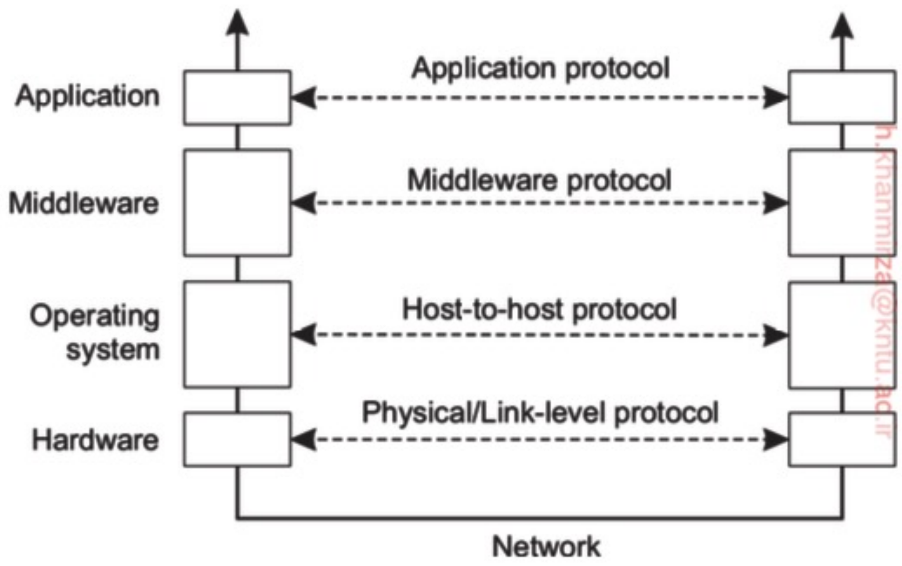
- Disadvantages
  - Proper dividing of sub-systems
    - Cross layer design
  - Duplicate functions
    - Error Checking in all layers

Layered Architecture

- Middleware is logically resides in Application layer
  - DNS
  - Distributed Locking
  - Distributed Commit
  - RPC (Remote Procedural Call)
  - Authentication

Layered Architecture

- Layered architecture in higher layer
  - Communication stack is embedded in OS
  - Middleware is like session & presentation layer in OSI model



Types of Communication

Persistent vs. Transient

- Persistent
  - The receiver need not be alive when the message is submitted.
  - A middleware takes the message from sender and does the job
  - Needs a middleware to store messages until delivery
  - Example: Mail System
    - Sender sends email, where Receiver may not running at that time

- Transient
  - Sender and receiver must be alive; otherwise message will be discarded
  - Example: all transport level communications are transient

Synchronous vs. Asynchronous

- Asynchronous
  - Sender continues to the next task after sending with no waiting
  - When reply is received, a callback routine is called
  - No assumptions about process execution speeds or message delivery times are made (unbounded)
- Synchronous
  - Sender blocks after sending until one of the following conditions:
    - A middleware takes the message
    - The message is sent and delivered to the intended recipient
    - The response is received from recipient
  - We assume execution speed and message-delivery time is bounded

Synchronous vs. Asynchronous

- Partial-Synchronous
  - We estimate a bound for process execution speed and message delivery
  - We use time-outs to conclude the other side has been crashed, but it can be false

Synchronous vs. Asynchronous

- Synchronous Model

Code blocks here

  - Print("Before");
  - String reply = send_msg(msg)
  - Print("Receiver said:" + reply);
  - Print("After")

- Asynchronous Model

Call Back function

  - Print("Before");
  - send_msgAsync(msg, reply_rcvd)
  - Print("After")
  - void reply_rcvd(String reply){
    - Print("Receiver said:" + reply);
  - }

```
Before
Receiver said: salam!
After
```

```
Before
After
Receiver said: salam!
```

Synchronous vs. Asynchronous

- Isochronous:
  - Communication has maximum and minimum end-to-end delays
  - jitter is bounded (streaming audio, video, sensor data)

- Stream-Oriented Communication
  - A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission
  - Stream types
    - Simple: consists of a single flow of data (e.g., audio or video)
    - Complex: multiple data flows (e.g., stereo audio or combination audio/video)

Combinations

- Persistent + Synchronous
  - Email
  - Message Queuing systems
- Transient + Synchronous
  - RPC

Remote Procedural Call (RPC)

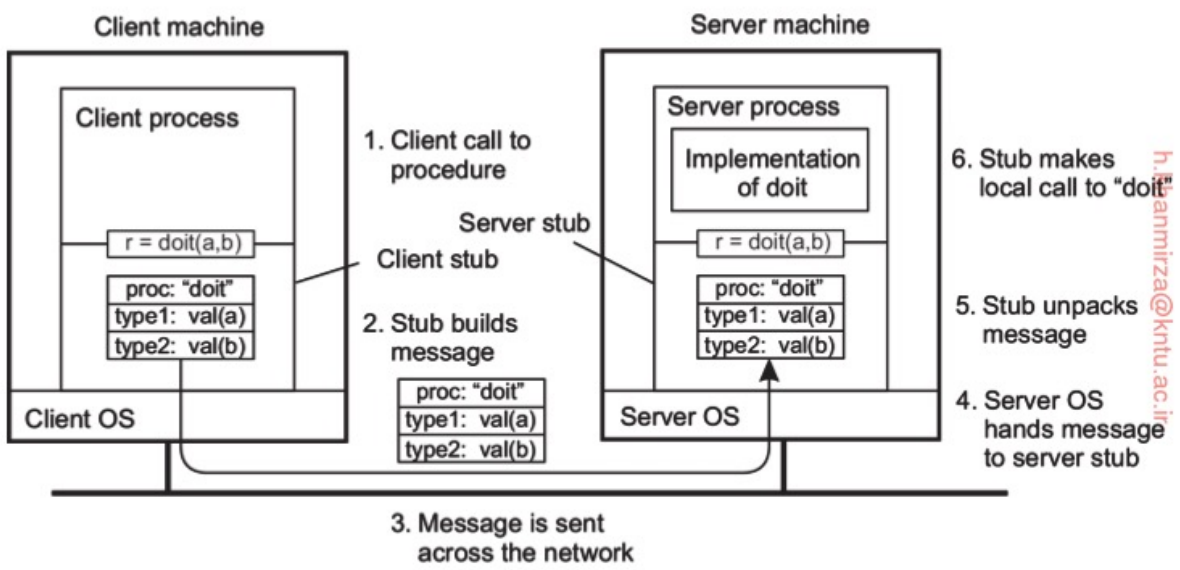
RPC

- Hiding Complexity
  - IP hides complexities in routing
  - TCP hides complexities in reliable communication

- RPC Scenario
    - Process A calls a function C from service on machine B
    - Process A is blocked until completion of the procedure and return of response
    - Function C is executed on machine B

- RPC hides complexities of explicit message exchange
  - RPC transparency, make it looks like a local procedure call

Remote Call - Client

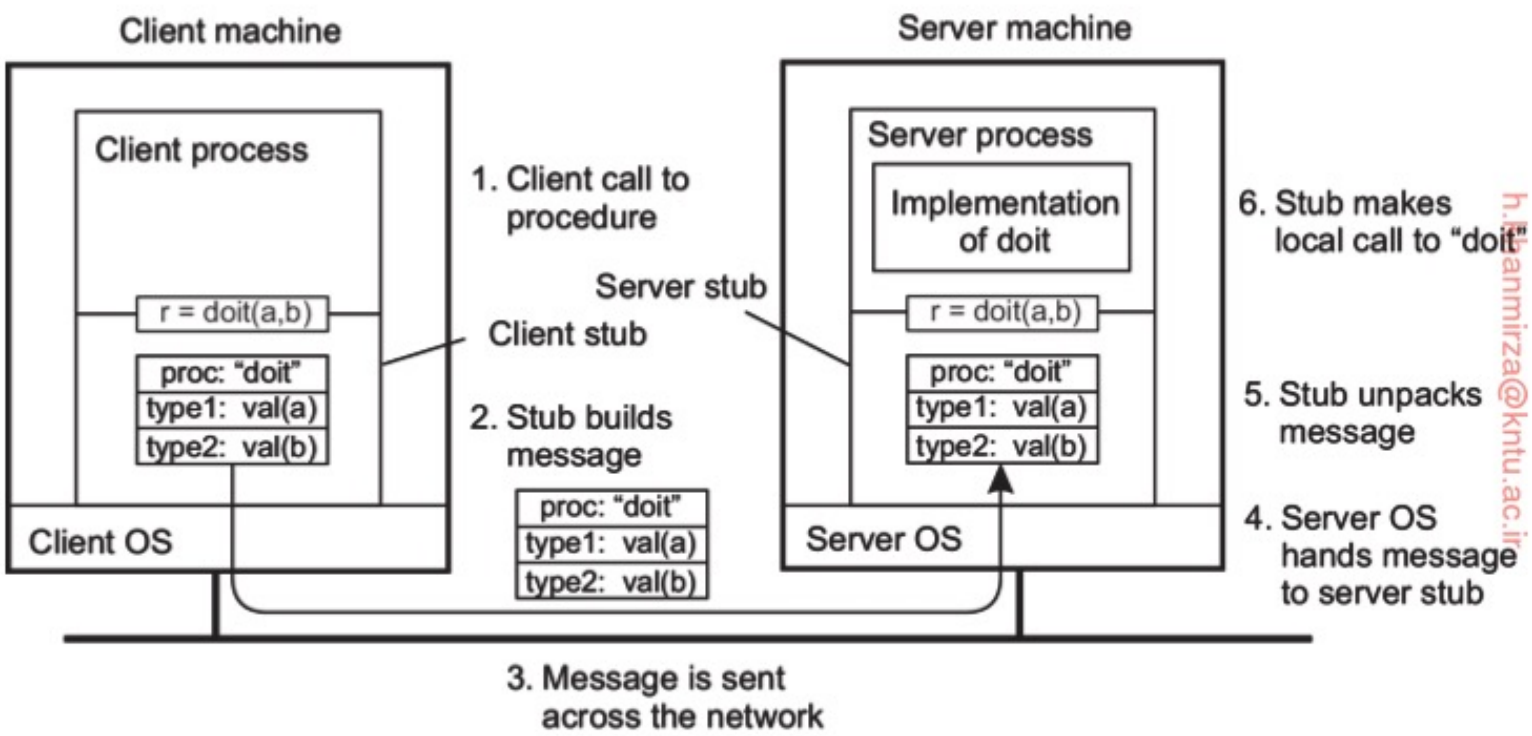


When a local call is done Linker loads the library file Finds the function offset and executes it

1. A Library called client stub is linked
2. Client stub has the same interface as the intended function but has no implementation
3. Client stub packs (marshals) variables into a message
4. Sends it to a RPC server on other machine

Remote Call - Server

1. In Server, server stub receives the message
2. Server stub unpacks (unmarshall) the message,
3. Loads and links the library then calls the intended function
4. Packs the result in a message
5. Sends back the result to client stub
6. Client stub returns the result to the caller which is blocked up-to-now



Simple Sample in Python

```
void main () {
    ….
    client.append("item", prevList)
    …
```

```
class DBList:
  def append(self, data):
    self.value = self.value + [data]
    return self

class Server:
  def append(self, data, dbList):
    return dbList.append(data)

  def run(self):
    while True:
      msgreq = self.chan.recvFromAny() # wait for any request
      client = msgreq[0]               # see who is the caller
      msgrpc = msgreq[1]               # fetch call & parameters
      if APPEND == msgrpc[0]:          # check what is being requested
        result = self.append(msgrpc[1], msgrpc[2]) # do local call
        self.chan.sendTo([client],result)          # return response
```

Same Interface

```
class Client:
  def append(self, data, dbList):
    msglst = (APPEND, data, dbList)          # message payload
    self.chan.sendTo(self.server, msglst)    # send msg to server
    msgrcv = self.chan.recvFrom(self.server) # wait for response
    return msgrcv[1]                         # pass it to caller
```

Client Stub

Server Stub

Simple Sample in Python

```
void main () {
    ….
    client.append("item", prevList)
    …
```

```
class DBList:
  def append(self, data):
    self.value = self.value + [data]
    return self
```

```
class Server:
  def append(self, data, dbList):
    return dbList.append(data)

  def run(self):
    while True:
      msgreq = self.chan.recvFromAny() # wait for any request
      client = msgreq[0]               # see who is the caller
      msgrpc = msgreq[1]               # fetch call & parameters
      if APPEND == msgrpc[0]:          # check what is being requested
        result = self.append(msgrpc[1], msgrpc[2]) # do local call
        self.chan.sendTo([client],result)          # return response
```

```
class Client:
  def append(self, data, dbList):
    msglst = (APPEND, data, dbList)          # message payload
    self.chan.sendTo(self.server, msglst)    # send msg to server
    msgrcv = self.chan.recvFrom(self.server) # wait for response
    return msgrcv[1]                         # pass it to caller
```

Client Stub

Server Stub

Some Languages generate stub code automatically

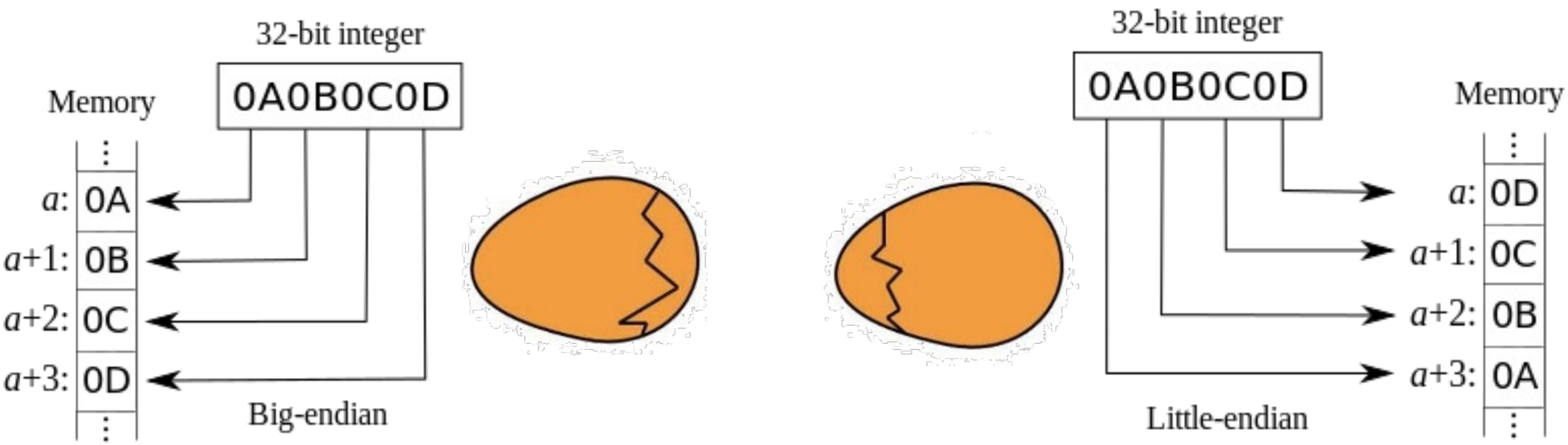
RPC

- RPC Problems
  - Parameter passing (pointers, objects)
    - Processes run on different address space
  - Failure

Parameter Passing

- Un/Marshaling
  - Un/Packing parameters in a message
- What happens if client and server machines has different platforms
  - Big Endian / Little endian format battle
  - Use machine- and network-independent format (normally Big-Endian!)

Parameter Passing: Big Endian vs Little Endian



Parameter Passing

- How about sending pointers and references?
  - A pointer is meaningful only within the address space of the process in which it is being used.

- Common local parameter passing models
  - Copy-By-Value func(int x) { x = 10;}
  - Copy-By-Reference func(int *x) { *x = 10}

- In case of RPC we may use the following strategy
  - Copy-By-Value/Restore

  - What is difference with Copy-By-Refrence?

Parameter Passing

- Solutions
  - Forbid pointers and references

  - Pointers and references must be used only for
    - Fixed-Size data structures
    - Dynamic-sized data structures if their size can be computed easily in run-time
      - Like String/dynamic arrays
      - Just do Copy-By-Value/Restore instead of Copy-By-Ref
    - It is good enough
    - What if the pointer points to a file, net socket, ⋯?

Remote Method Invocation (RMI)

- The first Object-Oriented RPC
- Calls methods of an object from another Java Virtual Machine (JVM)
- Language Support
  - Un/Packing is simpler just make objects implement Serializable Interface
  - Distributed Garbage Collection
    - Automatic collection of remote objects when no client using them

RMI Sample

```java
public interface RemoteServerInterface extends Remote{
	double computeOnDoubles(List<Double> numbers) throws RemoteException;
	int computeOnIntegers(List<Integer> numbers) throws RemoteException;
}
```

RMI Sample (server impl.)

```
public class RemoteServerInterfaceImpl extends UnicastRemoteObject implements
RemoteServerInterface
{
        protected RemoteServerInterfaceImpl() throws RemoteException {super(0);}
        public double computeOnDoubles(List<Double> numbers) {
                double sum = 0.0;
                for (Double number : numbers) {sum += number;}
                return sum;
        }
        public int computeOnIntegers(List<Integer> numbers) {
                int sum = 0;
                for (Integer number : numbers) {sum += number; }
                 return sum;
}}
```

RMI Sample (starting server)

```
public static void main(String[] args) {
        System.out.println("RMI server started");
        try {
                LocateRegistry.createRegistry(1099);
                System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
                System.out.println("java RMI registry already exists."); }
        try {
                RemoteServerInterfaceImpl obj = new RemoteServerInterfaceImpl();
                // Bind this object instance to the name "RmiServer“
                Naming.rebind("//localhost/RmiServer", obj);
                System.out.println("PeerServer bound in registry");
        } catch (RemoteException e) {        e.printStackTrace();    }
          catch (MalformedURLException e) {        e.printStackTrace();    }}
```

RMI Sample (client impl.)

```
public class RemoteServerClient {
  public static void main(String[] args) {
      try {
        RemoteServernIterface remoteServer =
            (RemoteServerInterface)   Naming.lookup("//localhost/RmiServer");

        List<Double> list = new ArrayList<Double>(Arrays.asList(1.0, 2.3));

        double result = remoteServer.computeOnDoubles(list);

        System.out.println("result = " + result);
      }
      catch (Exception e) {e.printStackTrace();}
}
```

RMI Sample

- Then you compile it with rmic.exe (RMI compiler)
  - It generates **RemoteServerInterfaceImpl_Stub** class
  - In newer versions, there is no need to skeleton (server stub) class

RMI Sample

Decompiled .class file, bytecode version: 45.3 (Java 1.1)

```java

    public double computeOnDoubles(List $param_List_1) throws RemoteException {
        try {
            Object $result = super.ref.invoke(this, $method_computeOnDoubles_0, new Object[]{$pa
            return (Double)$result;
        } catch (RuntimeException var3) {
            throw var3;
        } catch (RemoteException var4) {
            throw var4;
        } catch (Exception var5) {
            throw new UnexpectedException("undeclared checked exception", var5);
        }
    }

    public int computeOnIntegers(List $param_List_1) throws RemoteException {
        try {
            Object $result = super.ref.invoke(this, $method_computeOnIntegers_1, new Object[]{$pa
            return (Integer)$result;
        } catch (RuntimeException var3) {
            throw var3;
        } catch (RemoteException var4) {
            throw var4;
        } catch (Exception var5) {
            throw new UnexpectedException("undeclared checked exception", var5);
        }
    }
}
```

Generated Client Stub

RMI

- RMI generates all the code in the next slides
  - You had to write yourself

RMI

```java
import java.net.*;
import java.io.*;

public class client {

    public static void main(String[] args) {
        Socket client = null;
        ObjectOutputStream out = null;
        ObjectInputStream in = null;

        try {

            client = new Socket("127.0.0.1", 8888);
            out = new ObjectOutputStream(client.getOutputStream());
            in = new ObjectInputStream(client.getInputStream());

            Student student = new Student(96, "John");
            out.writeObject(student);
            out.flush();

            out.close();
            in.close();
            client.close();

        } catch (UnknownHostException e) {
            e.printStackTrace();
            System.exit(1);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }

    }
}
```

RMI

```java
import java.net.*;
import java.io.*;
import java.io.*;
public class server {

    public static void main(String[] args) {

        //The client is used to handle connections with a client once a connection is
        //established.
        Socket client = null;

        //The following two objects handles our Serialization operations, ObjectOutputStream
        //writes an object to the stream. ObjectInputStream reads an object from the stream.
        ObjectOutputStream out = null;
        ObjectInputStream in = null;

        try {
            ServerSocket server = new ServerSocket(8888);
            client = server.accept();
            out = new ObjectOutputStream(client.getOutputStream());
            in = new ObjectInputStream(client.getInputStream());

            Student student = (Student) in.readObject();
            System.out.println("Average: " + student.getStudentAvg() + " Name: " + student.getStudentName());

            // close resources
            out.close();
            in.close();
            client.close();
            server.close();

        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
```

RMI

- RMI mechanism works only in Java Virtual Machine

- Platform-Independent Mechanisms
  - CORBA
    - Is language independent RPC system based on objects
    - It defines IIOP, a protocol to exchange binary data between end points
      - RMI also supports IIOP
    - In CORBA you should define procedures using an IDL
    - It was very heavy weight and had vendor lock-in problem
  - Web Services
    - CORBA is killed by emergence of XML and Web Services
    - Objects can be serialized to XML, text-based readable format
    - IDL in Web Services is WSDL (Web Service Definition Language)

RPC Variations

- Asynchronous RPC
  - Client send request and waits for the server to accept the request, but does not wait for the result
- Multicast RPC for fault-tolerant communication
  - Sending to several servers
  - Just keep first reply, ignore subsequent replies

RPC Semantics in Failures

RPC Semantics in failures

1- Client is unable to locate the server

- Loss of transparency in case of reporting error!

RPC Semantics in failures

2- Request message from the client to the server is lost.

- Use timeout mechanism to re-send the request

- After several re-sends and getting no response, we are back to the 1st error

RPC Semantics in failures

3- Server crashes after receiving a request

- Just do the timeout trick

Exactly-Once Execution

- In some scenarios we want to execute a process exactly-once.
- Why this is important?
  - Idempotent requests vs. non-idempotent
    - Solving an equation
    - Depositing from a bank account

- It Is not clear for client when the failure has been occurred, both of them appears as a timeout

Exactly-Once Execution

- Suppose server crashes after receiving a request, could we provide a guarantee that a request will execute exactly once?

- To guarantee we do some extra actions in server
  - Server sends ACK after receiving request as a message delivery ack
  - Possible sever strategies
    - Send completion message before processing (M → P)
    - Send completion message after processing (P → M)

  - Possible time-out client strategies
    - Resend request
    - Give-up and report failure
    - Resend request when receive ack message
    - Resend request when not receive any ack message

Exactly-Once Execution

	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	Server			Server		

OK = Document processed once
DUP = Doc [illegible] processed twice
ZERO = Doc [illegible] not processed at all

Client Strategies

Server Strategies

Exactly-Once Execution

	Strategy M → P			Strategy P → M		
Reissue strategy	**MPC**	**MC(P)**	**C(MP)**	**PMC**	**PC(M)**	**C(PM)**
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	**Server**			**Server**		

OK = Document processed once
DUP = Document processed twice
ZERO = Document not processed at all

Exactly-Once Execution

	Strategy M → P			Strategy P → M		
Reissue strategy	**MPC**	**MC(P)**	**C(MP)**	**PMC**	**PC(M)**	**C(PM)**
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	**Server**			**Server**		

OK = Document processed once
DUP = Document processed twice
ZERO = Document not processed at all

No Guarantee for Exactly-Once

RPC Semantics in failures

4- Reply message from the server to the client is lost

- Client just sees timeout!

- Client can set a SeqNo for each message and set a retry flag for subsequent requests

- Server must maintain state for each client , to remember recent SeqNo and send a response to client

RPC Semantics in failures

5- Client crashes after sending a request.

- Orphan computation (no parent waiting for the result!)

- Orphan Extermination:
  - Keep log for each RPC in client stubs, in restart kill orphans!
    - Large log files
    - Orphans may start several other RPCs

- Reincarnation
  - Divide time to numbered epochs, on reboot broadcast new epoch to all servers, they kill orphans
    - No write in disk
    - Unreachable servers when report back the result, their result will have obsolete epoch

RPC Semantics in failures

5- The client crashes after sending a request (cont.)

- Expiration
  - Consider a Time T, for completion of each request. If job not finished client must explicitly request for further time.
    - When client crash and time T is passed, all computations are killed
    - Choosing right T can be tricky
- All solutions have problems
  - If orphan acquired lock of resources?
  - If orphan already updated some DB records?
  - If orphan started new computations on other servers, they may not be trackable

Message-Oriented Communication

Message-Oriented Communication

- Transport-level sockets
  - It is still very low-level
  - You need to code for any extra features beyond UDP and TCP
  - You may easily do mistakes
- RPC
  - Transient message-oriented communication
  - Good abstraction, but it is still low-level, we need higher abstractions

Message-Oriented Communication

- Message Queuing
  - Persistent message-oriented communication
  - Provides new common communication patterns

- Famous products
  - ZeroMQ
  - RabbitMQ
  - Apache ActiveMQ
  - .....

ZeroMQ

- A wrapper on Socket API
- Supports several underlying protocols
  - inproc: inter-thread shared memory
  - IPC: between processes
  - TCP
  - Pgm, and epgm: for multicast
    - ZeroMQ higher level sockets support one-to-many and many-to-one communication
- Setting up, and maintaining connections, protocol issues, ··· all is kept mostly under the hood

ZeroMQ

- ZeroMQ uses special paired higher level socket types
  - Each pair of socket types corresponds to a communication pattern
    - Request-Reply
    - Publish-Subscribe
    - Pipeline
    - Router-Dealer

ZeroMQ

- Request-Reply pattern
  - Client uses REQ socket type
  - Server uses REP socket type
  - And nothing!

```
context = zmq.Context()

p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
s  = context.socket(zmq.REP)    # create reply socket

s.bind(p1)                      # bind socket to address
s.bind(p2)                      # bind socket to address
while True:
  message = s.recv()            # wait for incoming message
  if not "STOP" in message:     # if not to stop...
    s.send(message + "*")       # append "*" to message
  else:                         # else...
    break                       # break out of loop and end
```

Server Code

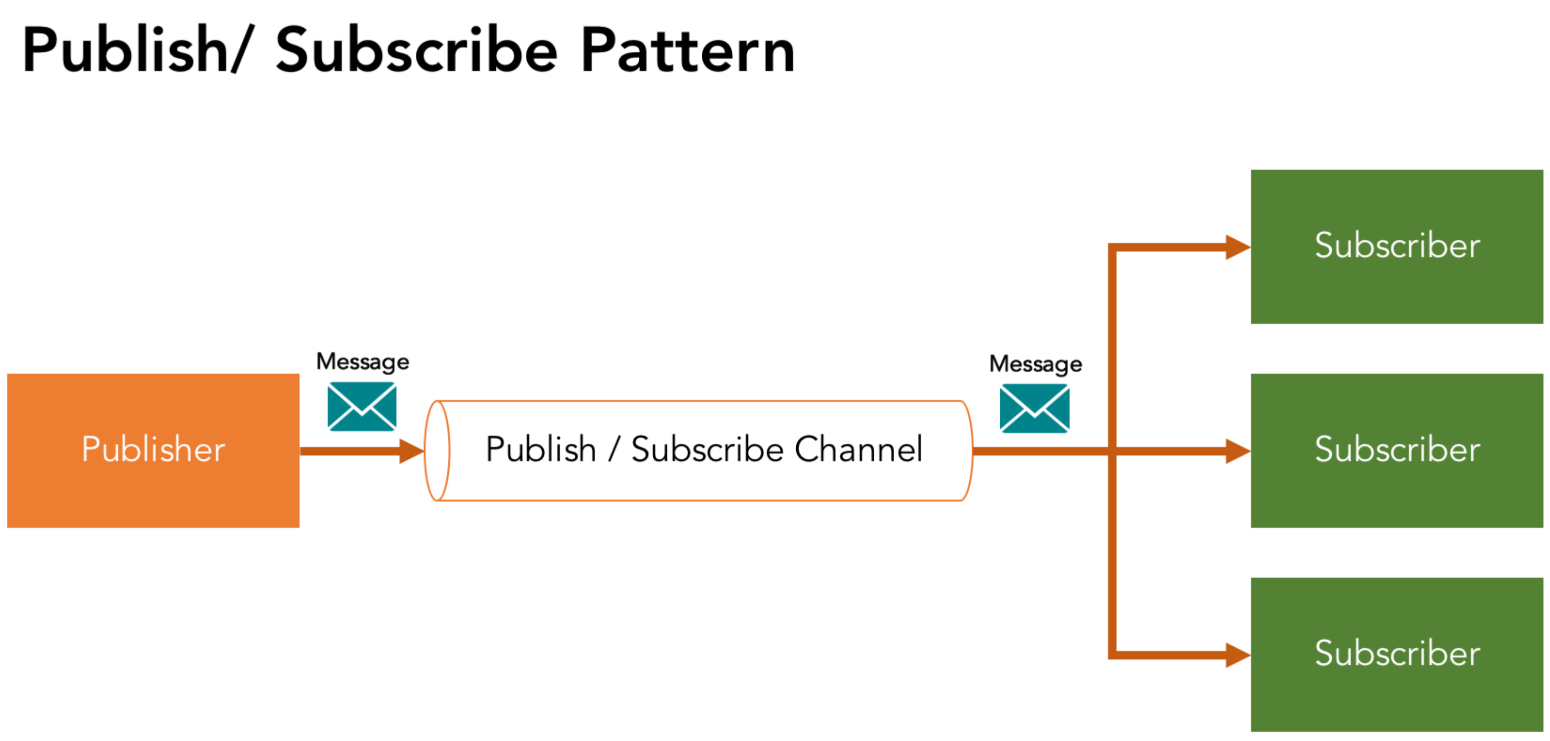
ZeroMQ

- Request-Reply pattern
  - Client uses REQ socket type
  - Server uses REP socket type
  - And nothing!

```
import zmq
context = zmq.Context()

p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
s  = context.socket(zmq.REQ)    # create request socket

s.connect(p1)                   # block until connected
s.send("Hello world 1")         # send message
message = s.recv()              # block until response
s.send("STOP")                  # tell server to stop
print message                   # print result
```

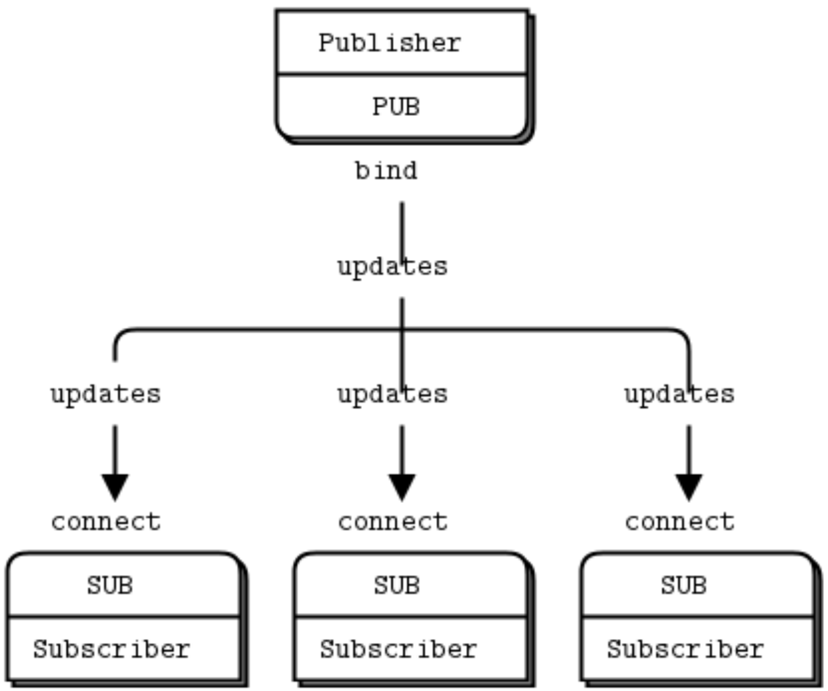
Client Code

ZeroMQ


Publish/ Subscribe Pattern
Publisher
Message
Publish / Subscribe Channel
Message
Subscriber
Subscriber
Subscriber

ZeroMQ

- Publish-subscribe pattern
  - An architectural style
  - Very useful to implement opennes
  - Clients subscribe to specific messages that are published by servers



ZeroMQ

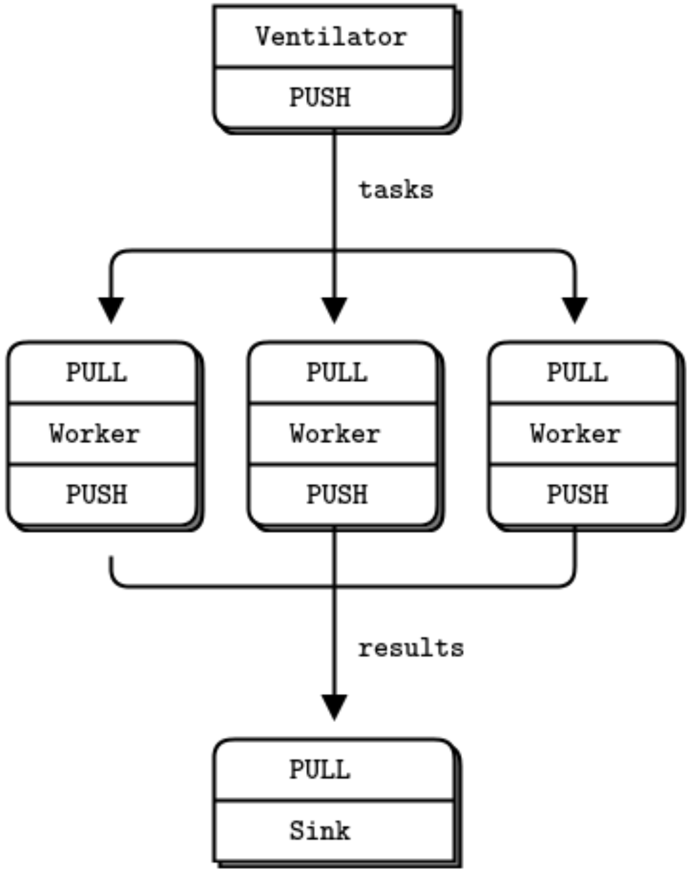
```
import zmq, time

context = zmq.Context()
s = context.socket(zmq.PUB)        # create a publisher socket
p = "tcp://"+ HOST +":"+ PORT      # how and where to communicate
s.bind(p)                          # bind socket to the address
while True:
  time.sleep(5)                    # wait every 5 seconds
  s.send("TIME " + time.asctime()) # publish the current time
```

```
context = zmq.Context()
s = context.socket(zmq.SUB)          # create a subscriber socket
p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
s.connect(p)                         # connect to the server
s.setsockopt(zmq.SUBSCRIBE, "TIME")  # subscribe to TIME messages

for i in range(5):  # Five iterations
  time = s.recv()   # receive a message
  print time
```

Server Code | **Client Code**

ZeroMQ

- Pipeline pattern
  - A process wants to push out its results, assuming that there are other processes that want to pull in those results
    - Pushing process has no care about which other process pulls in the results
  - One of the workers pulls the last result and do the computation task
  - Suitable for
    - Computation applied on a set of data
    - Parallel computation



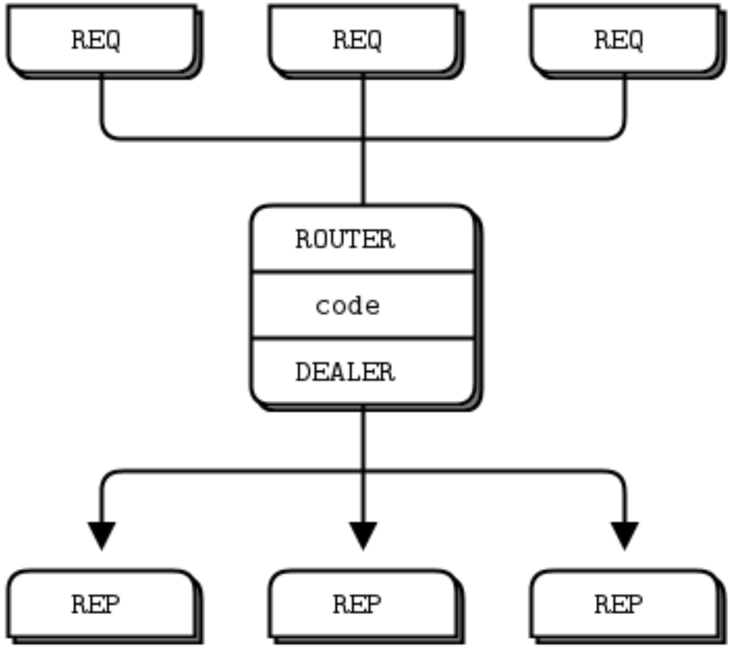
ZeroMQ

```
context = zmq.Context()
me  = str(sys.argv[1])
s   = context.socket(zmq.PUSH)        # create a push socket
src = SRC1  if me == '1' else SRC2    # check task source host
prt = PORT1 if me == '1' else PORT2   # check task source port
p   = "tcp://"+ src +":"+ prt         # how and where to connect
s.bind(p)                             # bind socket to address

for i in range(100):                  # generate 100 workloads
  workload = random.randint(1, 100)   # compute workload
  s.send(pickle.dumps((me,workload))) # send workload to worker
```

```
context = zmq.Context()
me = str(sys.argv[1])
r  = context.socket(zmq.PULL)       # create a pull socket
p1 = "tcp://"+ SRC1 +":"+ PORT1     # address first task source
p2 = "tcp://"+ SRC2 +":"+ PORT2     # address second task source
r.connect(p1)                       # connect to task source 1
r.connect(p2)                       # connect to task source 2

while True:
  work = pickle.loads(r.recv())     # receive work from a source
  time.sleep(work[1]*0.01)          # pretend to work
```

ZeroMQ

- Router-Dealer Sockets
  - Proxy pattern
  - Broker Pattern
- Advantage
  - No need for explicit connection of pairing sockets
  - Dynamic connection
    - Easy to add/remove nodes
  - Load Balancing
  - Routing Capabilities (general pub/sub pattern)
  - Non-blocking
- Disadvantage
  - An intermediate point needs higher memory and adds delay



Advanced Message Queuing Protocol

- AMQP
  - Open application-layer standard for exchanging messages
  - Defines binary wire-level protocol (No standard API)
    - Description of the format of the data that is sent across the network as a stream of bytes
  - Products can be compliant to this standard
    - Apache ActiveMQ
    - RabbitMQ
    - Azur ServiceBus

Higher Abstractions…

Message Passing Interface (MPI)

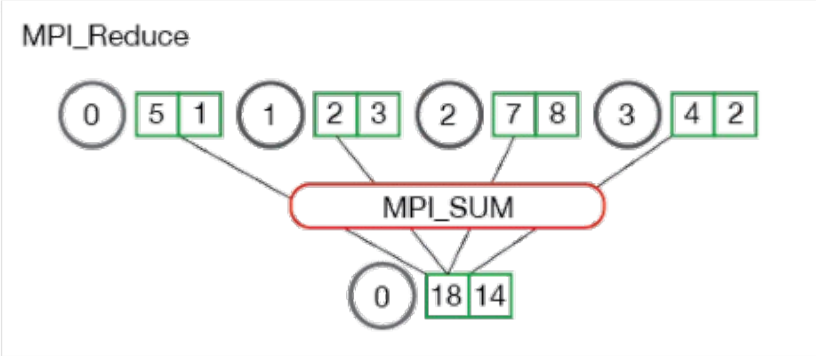
- A higher level than sockets with minimal overhead
  - Sum, Reduce, ⋯
  - Runs on IP with a fast efficient protocol for server clusters
  - Assumes serious failures such as process crashes or network partitions are fatal and do not require automatic recovery

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

MPI: Message Passing Interface

- Why so many different forms of communication
  - It gives users of MPI systems enough possibilities for optimizing performance
- MPI v3 has 440 operations!

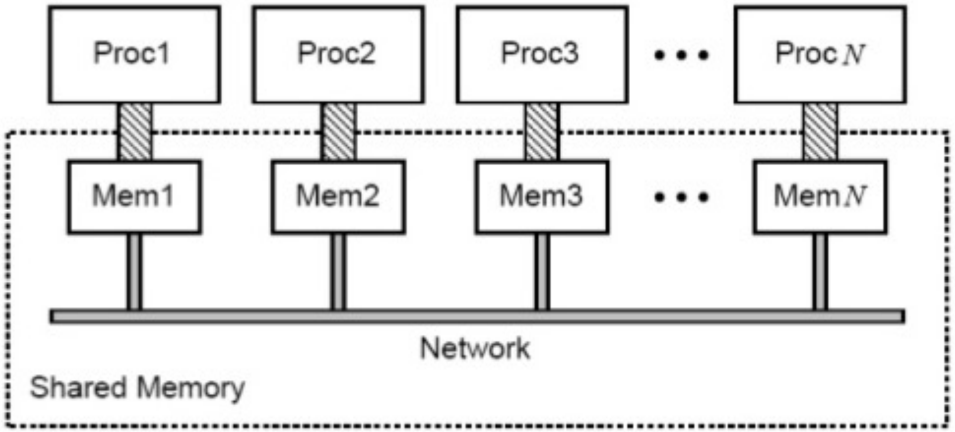
MPI Sample




```c
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) {
       local_sum += rand_nums[i];
}
// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

if (world_rank == 0) {  //if this is the root process
       printf("Total sum = %f, avg = %f\n", global_sum,
       global_sum / (world_size * num_elements_per_proc));
}
```

Distributed Shared Memory (DSM)

- Highest possible abstraction!
- DSM provides the illusion of shared memory on a distributed system
- DSM Purpose
  - Programmer writes parallel program: threads, shared variables, locks
  - DSM system farms out threads to a cluster of machines



DSM

- Why we need DSM
  - No change need for existing codes and libraries (= transparency)
  - Familiar model for developers
  - General purpose (vs. MapReduce, Spark)

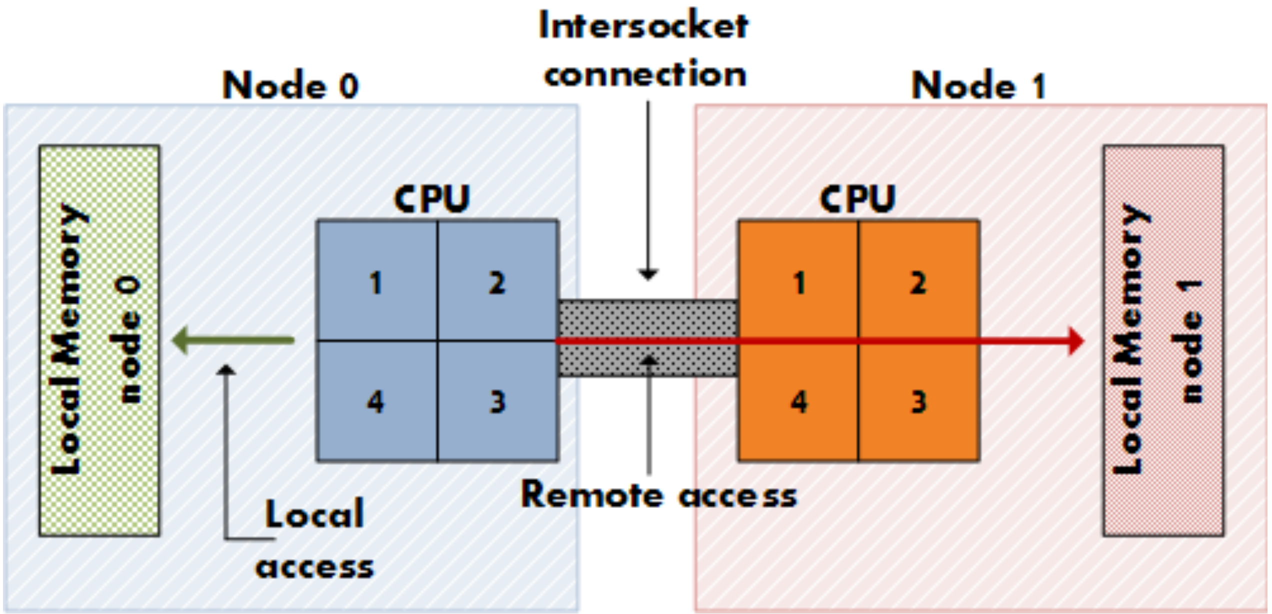
- DSM high level scenario
  - DSM runtime sends updates in messages between computers
  - Each computer has a local copy of recently accessed data items stored in DSM, for speed of access (cache)
  - DSM hides complexity of writing and dealing with message passing
  - Main problem is managing replicated data (efficient consistency)

DSM

- Very similar to NUMA architecture
  - However, DSM usually is implemented in LAN
    - In multi-processor system, processors have very high speed connection to remote memories
  - DSM is working over network, has to read from/write to another system
    - We need messaging for accessing remote memories

- It is very hard to implement an efficient real DSM!
  - We skip implementation models!

DSM



NUMA Architecture

DSM vs. Messaging

- DSM
  - Variables are accessed directly
    - Some implementations use built-in virtual memory and page fault mechanisms
    - Some implementations access variables with name
    - Needs other mechanisms like distributed locks, ..
  - Processes can have persistent communication
  - Measuring efficiency is very difficult
    - DSM is implicit and greatly depends on patterns of data sharing and data access
- Messaging
  - Marshaling/Unmarshaling
  - Persistent communications needs an extra entity
  - Efficiency can be measured explicitly in message-passing

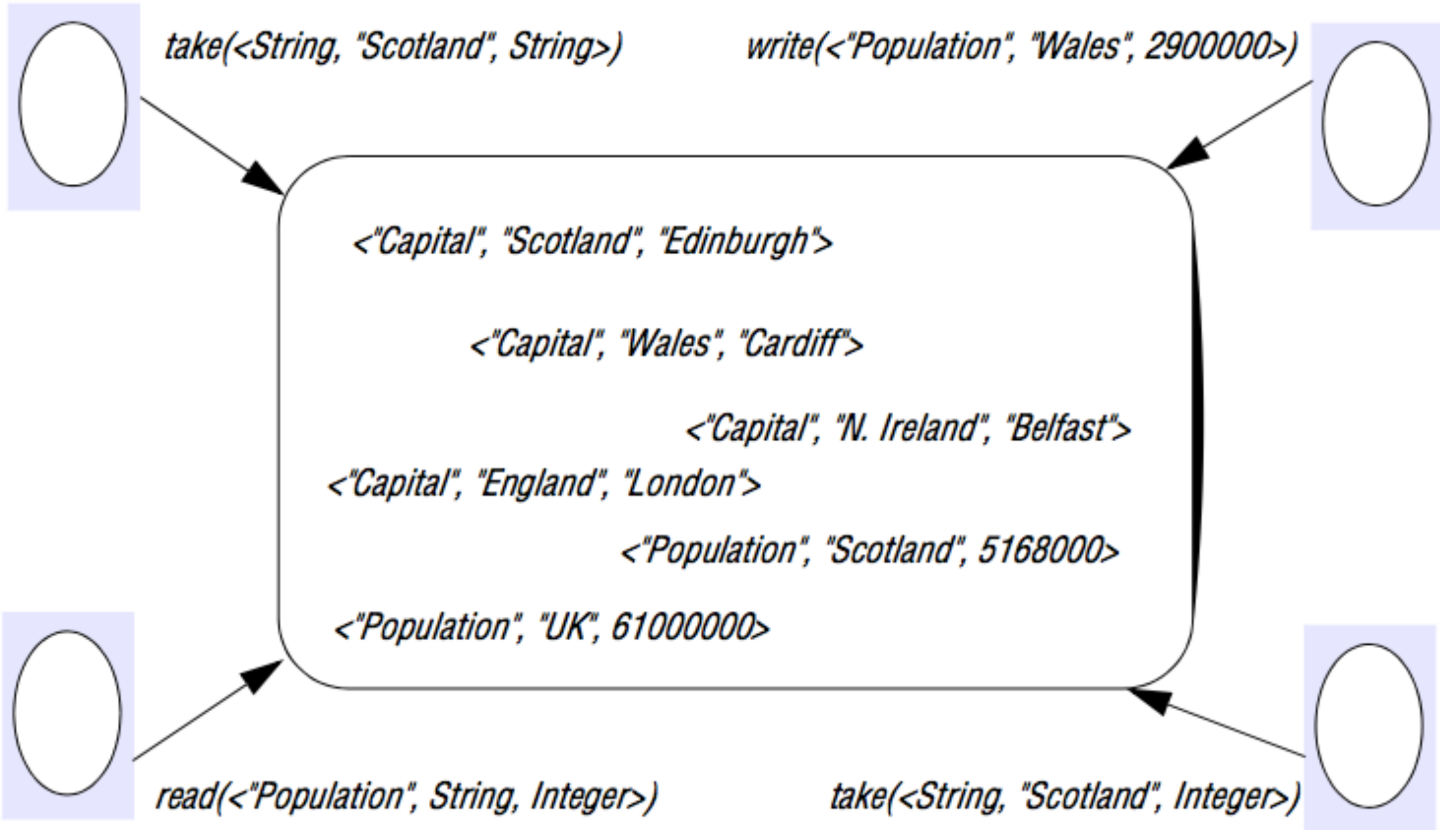
Tuple Spaces

- A form of DSM or content-based publish-subscribe system
- Processes communicate indirectly by placing tuples in a tuple space
- Several processes can read or remove tuples.
- Tuples do not have an address but are accessed by pattern matching on content
- Tuples spaces model has created Linda programming model
  - Vs. message passing, MapReduce, Spark, ···

Tuple Spaces

- Terminology
  - Tuple: consists of a sequence of one or more typed data fields:
    - `<“DS course", 1396>, <“StudentId", 987654>, <4, 9.8, "Yes">`
  - Tuple space: a shared collection of tuples
  - Processes share data by accessing the same tuple space
- Operations
  - Write: put tuple into a tuple space
  - Read: reads tuple
  - Take: removes the tuple from tuple space and keeps a local copy
  - Notify: when a tuple updated, inform listeners

Tuple Spaces



Tuple Spaces

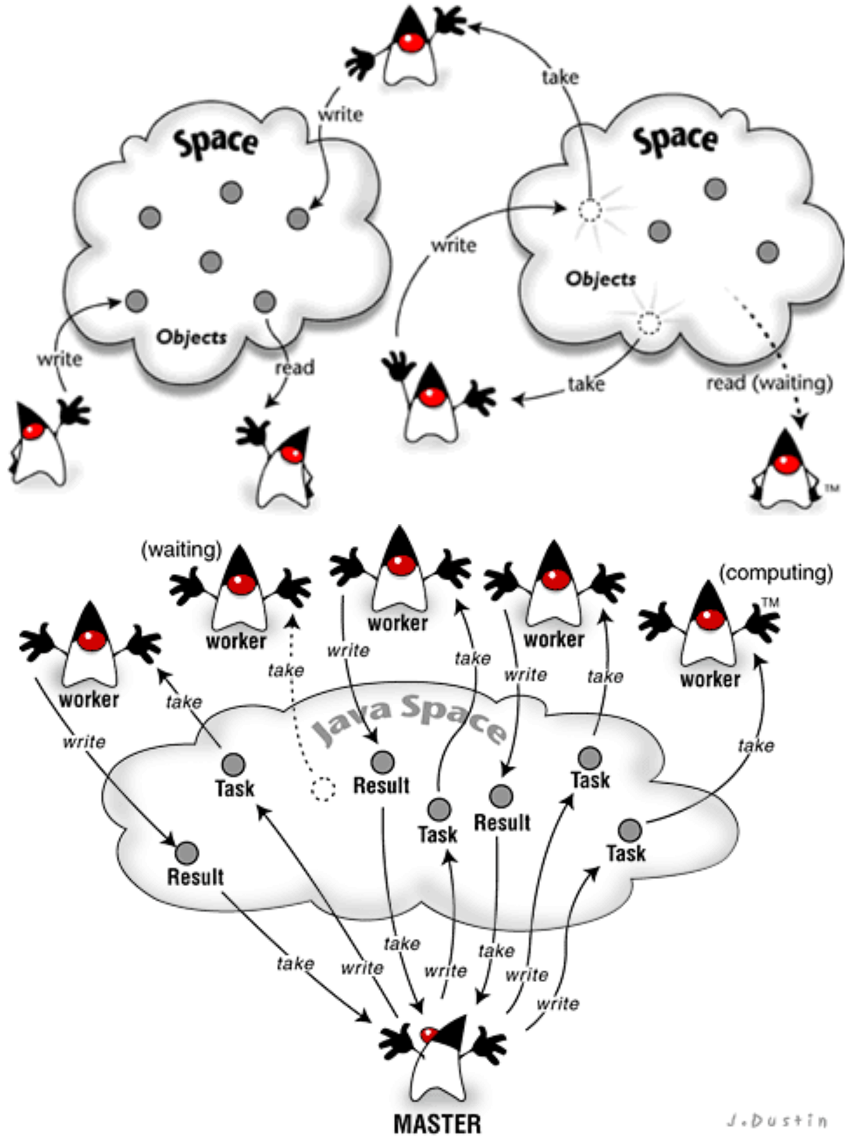
- Modification cannot be done in-place
  - Take the tuple
  - Write modified tuple
  - Example
    - `<s, count> := myTS.take(<"counter", integer>);`
    - `myTS.write(<"counter", count+1>);`
- Processes can register to receive notification when a particular tuple is updated

Tuple Spaces

- Variations
  - Having multiple Tuple Spaces (like variable scoping in {})
  - Replicate tuple space in several machines (consistency problem)
  - Put each tuple space in just one machine
    - How to find the intended tuple?
  - P2P models
- Implementations
  - JavaSpaces (high-throughput, low-delay, no fault-tolerant)
  - GigaSpaces: clustered & fault tolerant
  - Linda in a Mobile Environment (LIME): for mobile environments, needs no server

Tuple Spaces

- Java Spaces adds notify + transaction features
  - Easily build a chat server
  - Easily build a computation server
  - Easily build a distributed coordination service



Multicast Communication

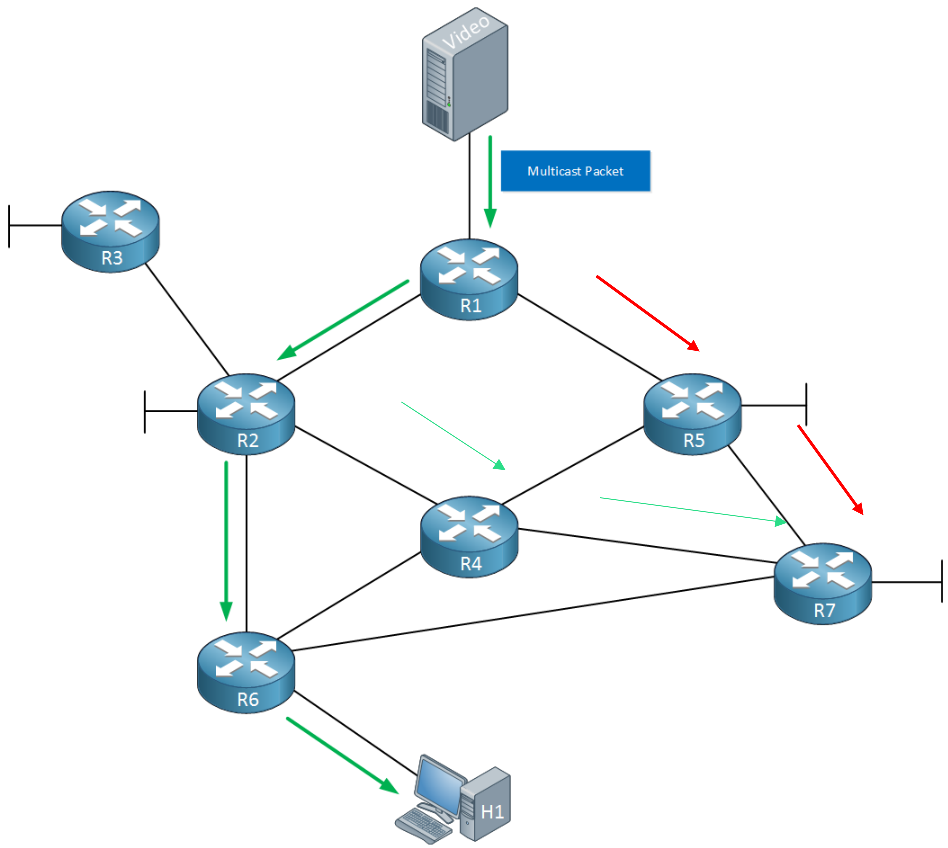
one-to-many communications

Multicast Communication

- Sending one data to several receivers
- At first, it was considered at lower-layer protocols, (Layers 2, 3, 4)
  - How to set-up minimum data distribution tree to deliver data

Multicast Tree

- Hosts cooperatively construct a tree
  - Efficient trees are Minimum Spanning Tree (MST) or a structure containing MST
  - It needs support of intermediate devices (routers, switches)
  - Huge management efforts



Multicast Communication

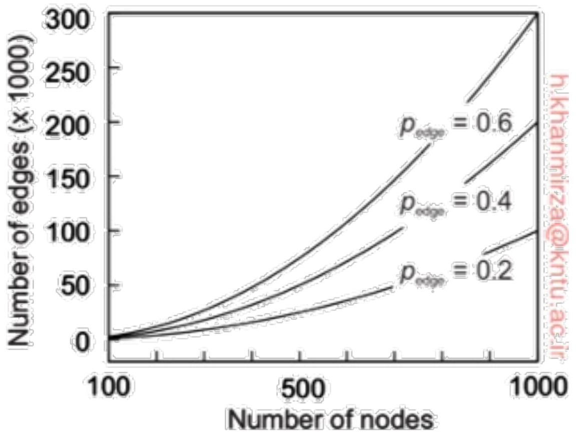
- With emergence of peer-to-peer applications, these protocols mostly implemented in application layer, as overlay networks
  - Implementation is easy in application layer
  - No need to interfere with network core
  - But, it may not be efficient

Broadcast

- Broadcast is a special case of Multicast
- A key design factor:
  - Minimize the use of intermediate nodes for which the message is not intended.
    - In multi-level trees, only the leaf nodes are the recipient of the message, others are mostly forwarders!
- Build a tree for each multicast group
  - Nodes belonging to several groups are in trouble!
    - Keep several list
    - Managing higher traffic

Broadcast

- Just flood a message in a multicast group in overlay network!
  - Very inefficient
  - Consider graph G(N,M), flooding means sending ≈ M messages
    - In tree, M = N -1 messages, the most efficient structure
    - In complete graph → $\binom{N}{2} = \frac{1}{2}N(N-1)$ messages
  - For other graphs we can have a estimate with Random Graphs or Erdös-Rényi graph.
    - Probability of having a link between two nodes is p_{edge}.
    - No of links ~ No of messages ~ $\frac{1}{2}p_{edge}N(N-1)$



Broadcast

- How to reduce flooding load?
  - Each node forward the message with p_{flood}
    - The total no. of messages drops linearly with this probability
    - If p_{flood} is small, some nodes may not receive the message
    - The probability of not receiving for a node with n neighbors is $(1 - p_{flood})^n$
  - Enhancement
    - Instead of an static probability use degree-dependent one

Gossiping

- Also known as Epidemic Behavior
  - Theory of epidemics studies the spreading of infectious diseases.
- While health organizations use this theory to prevent spreading infection, distributed system designers use the theory "infect" all nodes with new information as fast as possible.
- Has several variations
- Main theme is periodically send update data to random targets

Gossiping

- Terminology
  - A node with the data → infected
  - A node has not seen the data → susceptible
  - A node has data, but can not or will not spread the data → removed

Gossiping

- Anti-entropy propagation model:
  - P choose a neighbor Q at random and exchanges data with
  - At the next round, Q do the same with its neighbor

Gossiping

- Anti-entropy propagation model (cont.)
- Data exchange models
  - P pushes new updates to Q
    - Push-only models may not work correctly,
      - if many nodes are infected, the probability of selecting a susceptible node is small
    - Send multiple messages?
  - P pulls new updates from Q
    - Updates are triggered by susceptible nodes
    - Examine random subset of nodes to get possible updates
    - Having only 1 infected node, all nodes get infected
  - P and Q send updates to each other (push-pull)

Gossiping

- Gossiping Advantages
  - lightweight in large groups
  - Spreads a multicast message very quickly
  - Highly fault-tolerant

Gossiping

- If p_i is the probability that node P have not received the update in the i^{th} round then
- In pull-only model
  - $p_{i+1} = p_i \times p_i = p_i^2$
    - For P to remain susceptible in $(i+1)^{th}$ period, it must not receive in i^{th} round and in $(i+1)^{th}$ round
  - If p is small, then it converges rapidly to 0

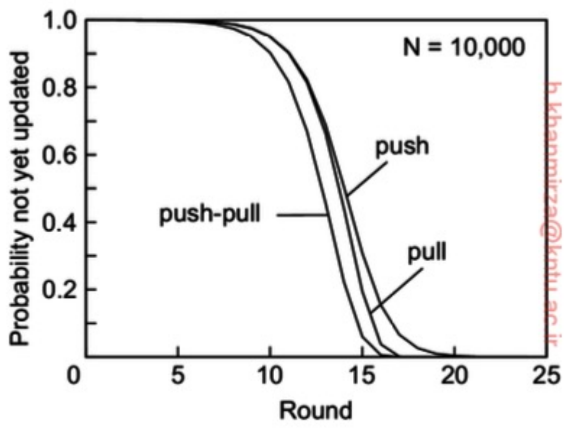
Gossiping

- If p_i is the probability that node P have not received the update in the i^{th} round then
- In push-only model
  - Average (expected) no. of updated nodes in i^{th} round: $n(1-p_i)$
  - Probability of sending update to P by any of nodes: $\frac{1}{n-1}$ → not sending: $1-\frac{1}{n-1}$
  - Then $p_{i+1} = p_i\left(1-\frac{1}{n-1}\right)^{n(1-p_i)}$
  - For small $p \ll n \Rightarrow p_{i+1} = p_i e^{-1}$
    - With this probability one node may not receive data at (i+1)th round

Gossiping

- Pull model is faster
- Push-pull model is faster → just combine both methods

- Figure shows how quickly the probability of not yet being updated drops as a function of the number of rounds.
  - Assuming nodes are up and running all the time, anti-entropy is an extremely effective dissemination protocol



The probability of not yet having been updated as a function of the number of dissemination rounds.

Rumor Spreading

- A variant of Gossiping
  - Node P has just been updated for data item x
  - It contacts other node Q, randomly and pushes the update to Q
  - If Q was already updated by another node, in that case, P may lose interest in spreading the update any further, with probability p_{stop}.
- Fast, but cannot guarantee that all nodes get the update.

Gossiping

- Gossiping is scalable
  - No need for synchronization
- Considering topology yields better results
  - Send update to nodes having less neighbors in common → directional gossiping
- In practice, a node may not know all members

The End!