LECTURE 6: MESSAGE-ORIENTED COMMUNICATION II: MESSAGING IN DISTRIBUTED SYSTEMS

Lecture Contents

- Middleware in Distributed Systems
- Types of Distributed Communications
  - Remote Procedure Call (RPC):
    - Parameter passing, Example: DCE
    - Registration & Discovery in DCE
  - Message Queuing Systems:
    - Basic Architecture, Role of Message Brokers
      - Example: IBM Websphere
    - Advanced Message Queuing Protocol (AMQP)
      - Example: Rabbit MQ
  - Multicast Communications:
    - Application Layer Messaging
    - Epidemic Protocols

SECTION 6.1: MIDDLEWARE IN DISTRIBUTED SYSTEMS

Role of Middleware

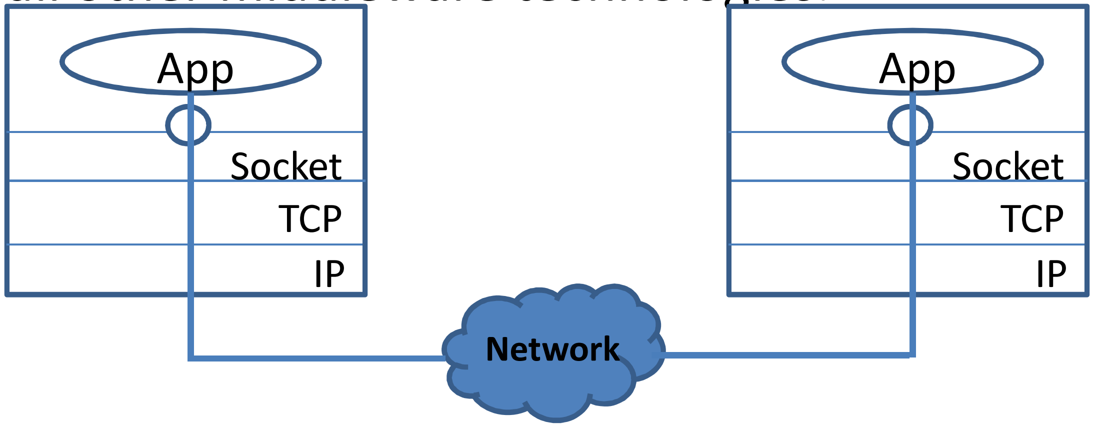
- *Observation*
  - Role to provide common services/protocols in Distributed Systems
  - Can be used by many different distributed applications
- *Middleware Functionality*
  - (Un)marshalling of data: necessary for integrated systems
  - Naming protocols: to allow easy sharing, discovery of resources
  - Security protocols: for secure communication
  - Scaling mechanisms, such as for replication & caching (e.g. decisions on where to cache etc.)
  - A rich set of comms protocols: to allow applications to transparently interact with other processes regardless of location.

Classification of Middleware

- Classify middleware technologies into the following groups:

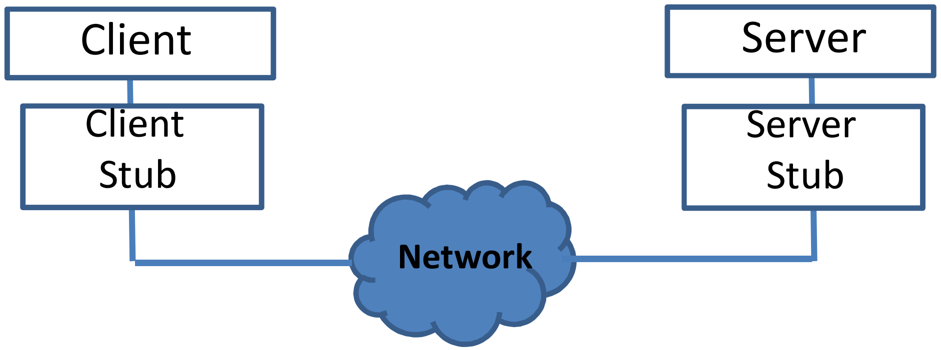
1. Bog-standard Sockets

- The basis of all other middleware technologies.



2. RPC – Remote Procedure Call (more later)

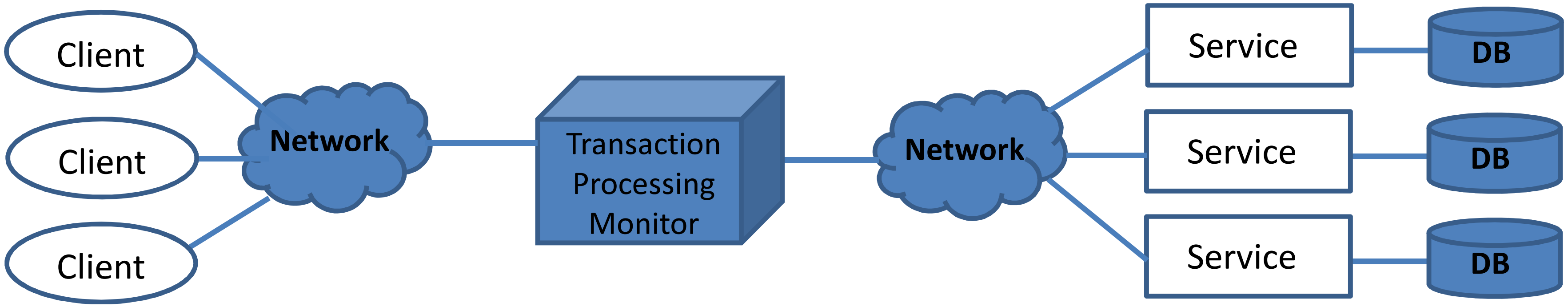
- RPCs provide a simple way to distribute application logic on separate hosts



Classification of Middleware (/2)

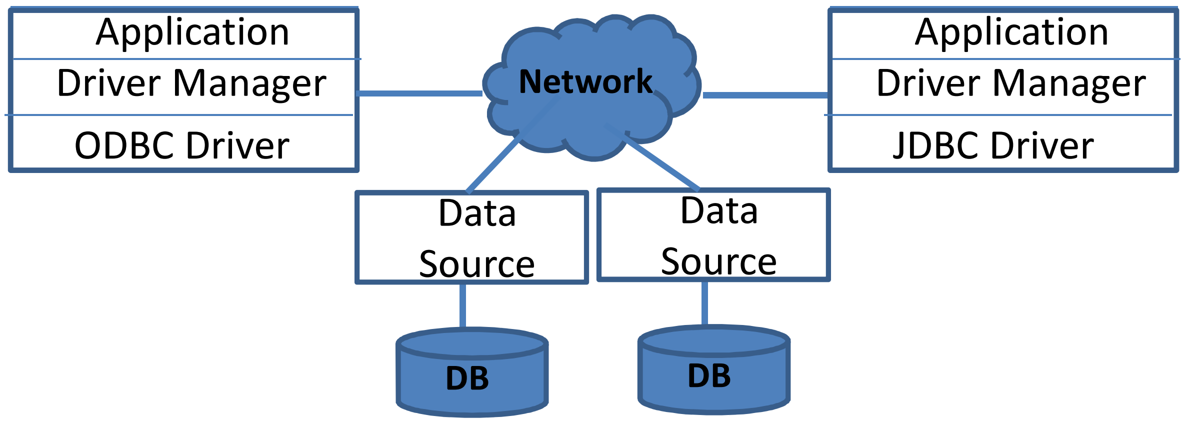
3. TPM - Transaction Processing Monitors:

- TPMs are a special form of MW targeted at distributed transactions.



4. DAM - Database Access Middleware:

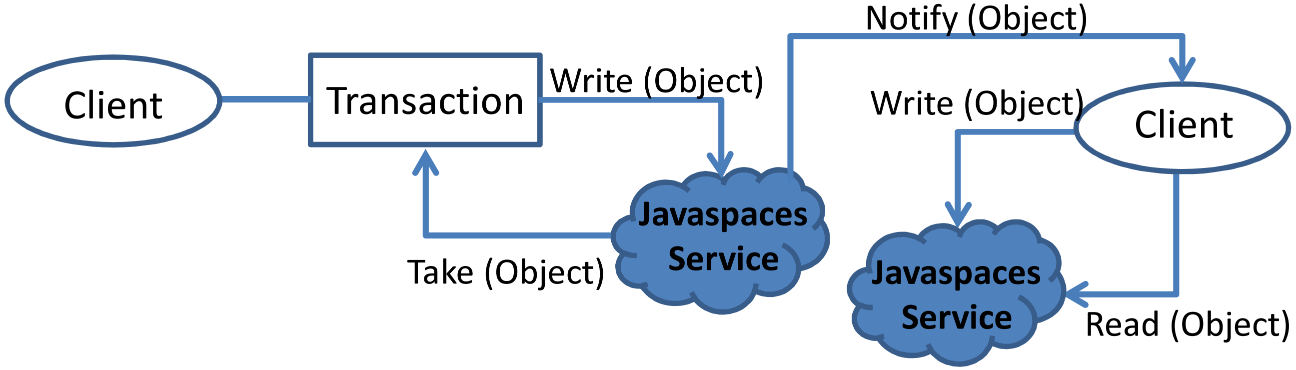
- DBs can be used to share & communicate data between distributed applications.



Classification of Middleware (/3)

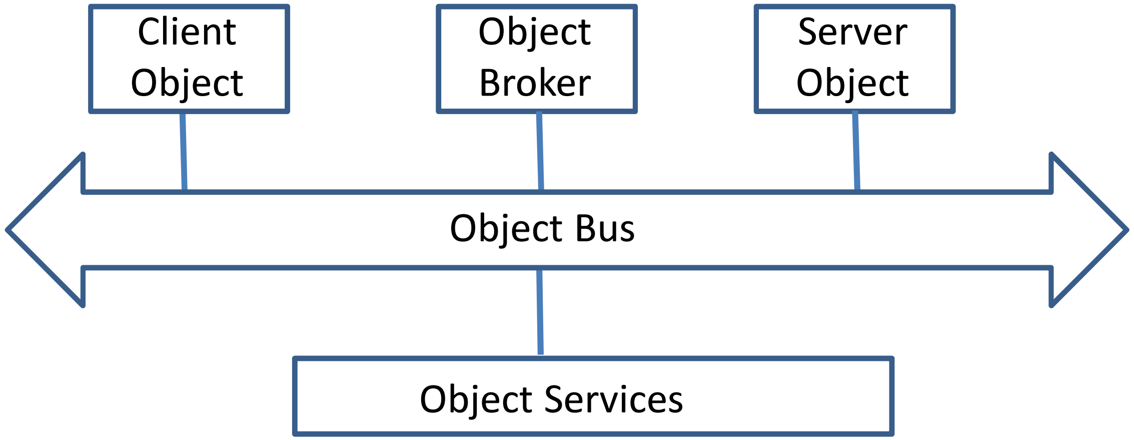
5. *Distributed Tuple:*

- Distributed tuple spaces implement a distributed shared memory space.



6. *DOT (Dist Object Technology) / OOM (Object-Oriented M/w):*

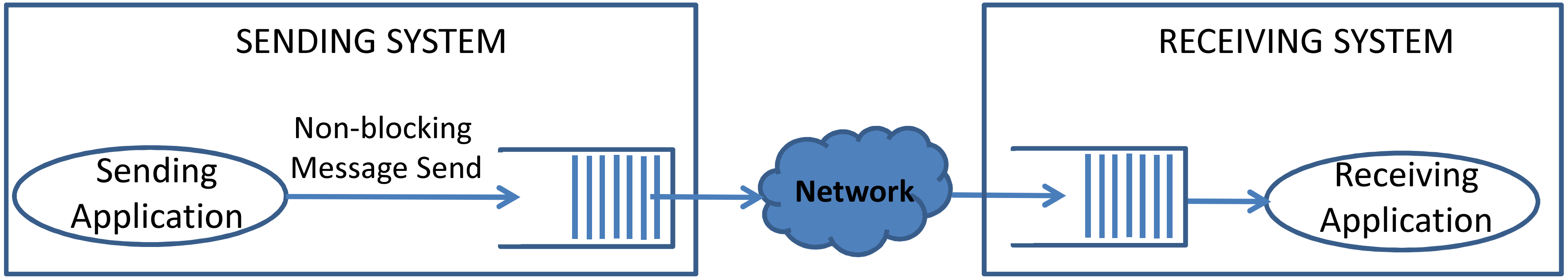
- DOT extends the object-oriented paradigm to distributed applications.



Classification of Middleware (/4)

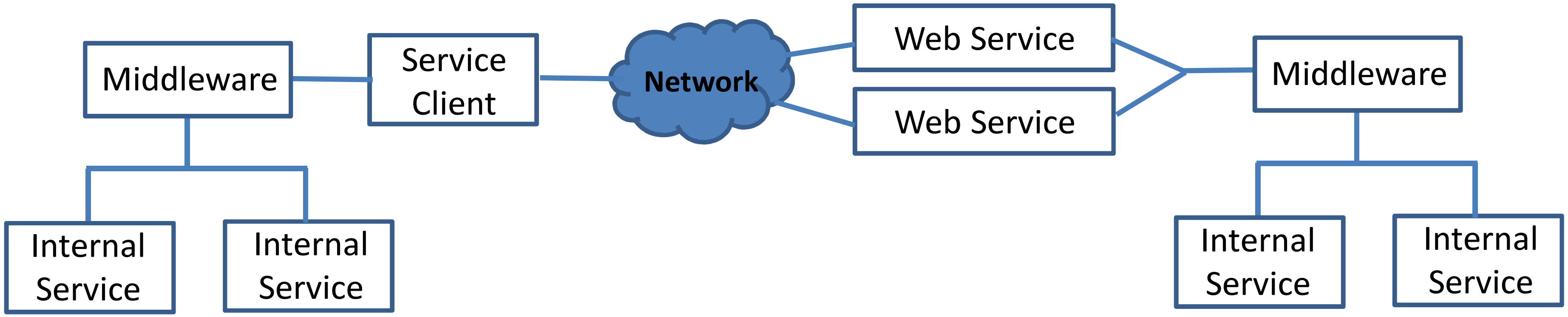
7. *MOM (Message Oriented Middleware):*

- In MOM, messages are exchanged asynchronously between distributed applications (senders and receivers).



8. *Web services:*

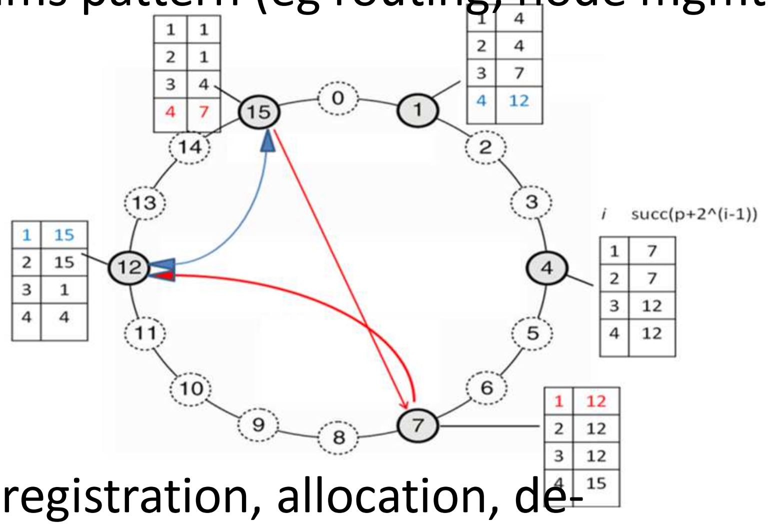
- Web services expose services (functionality) on a defined interface, typically accessible through the web protocol HTTP.



Classification of Middleware (/5)

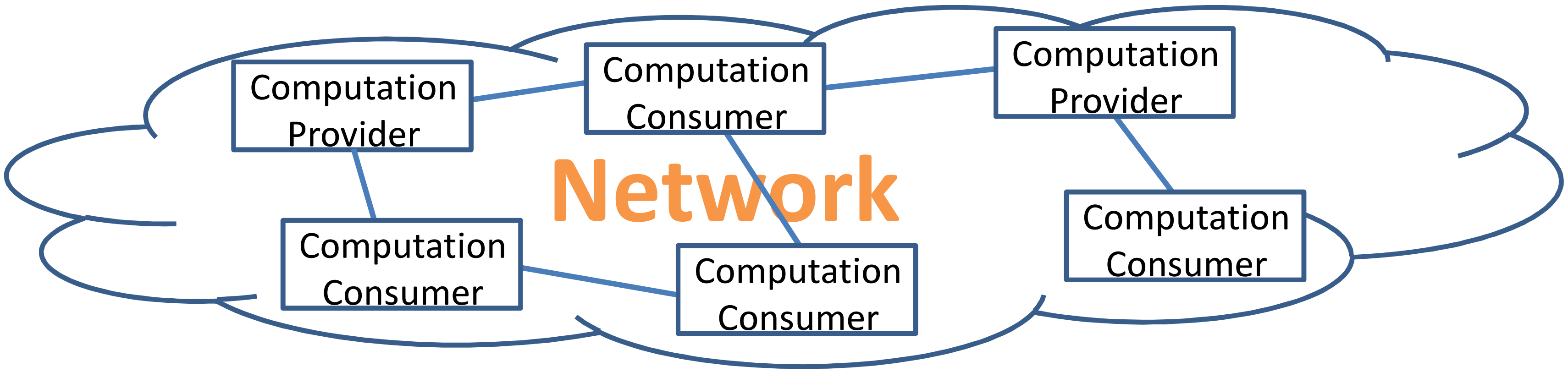
9. Peer-to-peer middleware:

- Have seen above how MW often follows particular *architectural style*.
- In P2P, each peer has equal role in comms pattern (eg routing, node mgmt)
- More on this later…



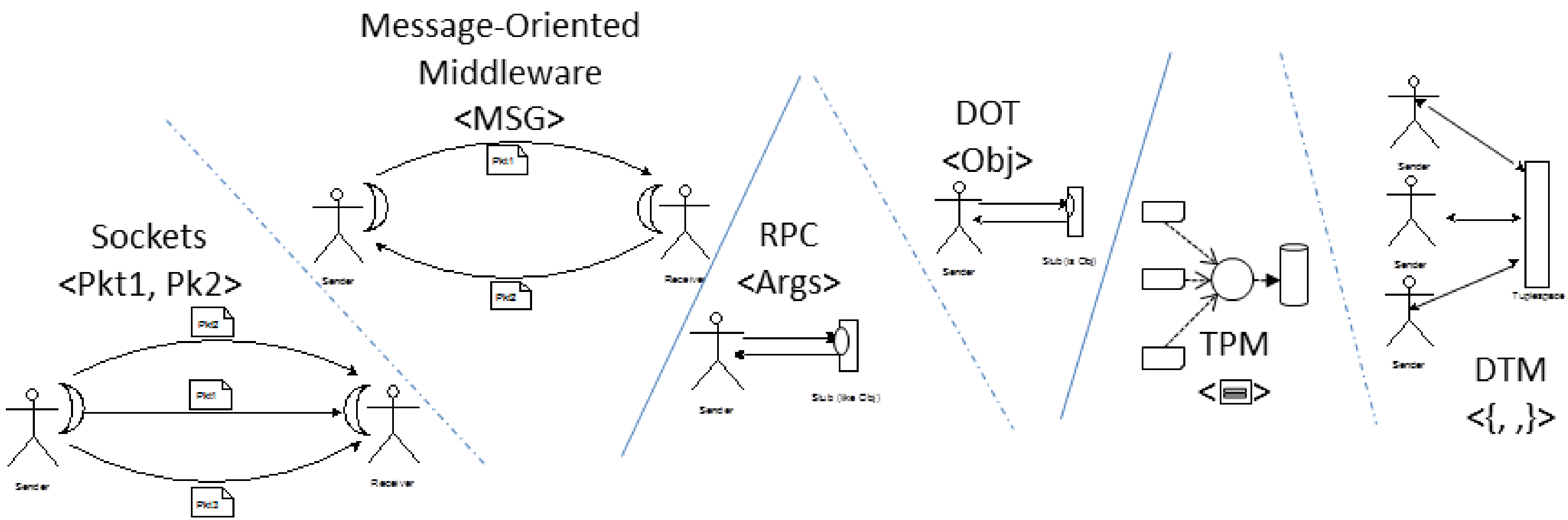
10. Grid middleware:

- Provides computation power services (registration, allocation, de-allocation) to consumers.



Summary of Communications Middleware

- Essentially a range of types of communications middleware
- All can be used to implement others, all are suited to different cases
  - All carry some payload from one side to another <with details>
  - Some of these payloads are ‘active’ and some are ‘passive’
  - Also differ in granularities and whether synchronous or not.



SECTION 6.2: COMMUNICATION IN DISTRIBUTED SYSTEMS

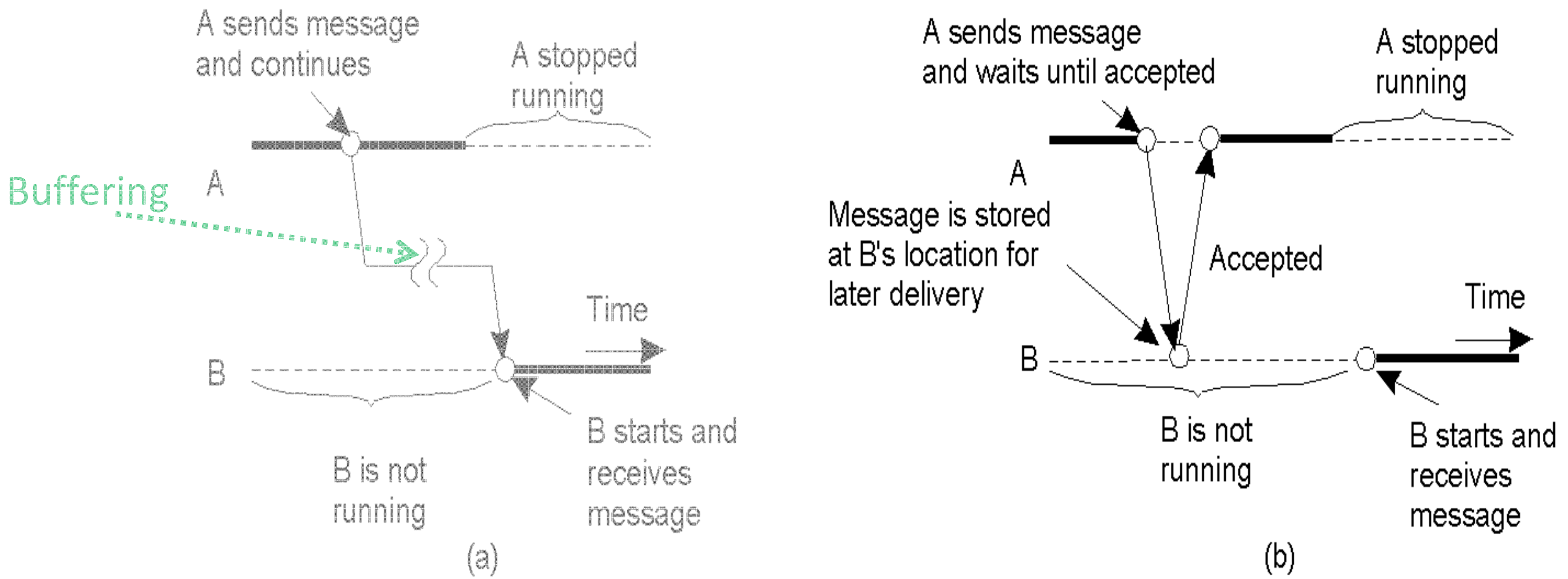
Terminology for Distributed Communications

- *Terminology for Distributed Communications*
  - *Persistent Communications:*
    - Once sent, the "sender" stops executing.
    - "Receiver" need not be in operation – communications system buffers message as required until delivery can occur.
  - *Transient Communications:*
    - Message only stored as long as "sender" & "receiver" are executing.
    - If problems occur either deal with them (sender is waiting) or message is simply discarded …

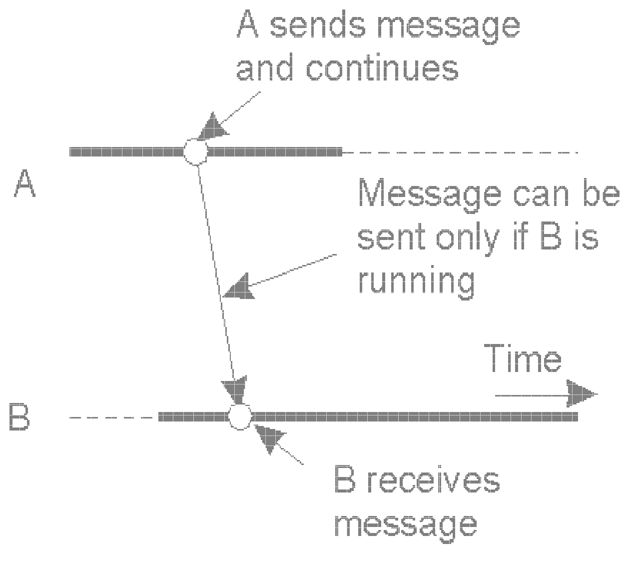
Persistence & Synchronicity in Communications

a) Persistent asynchronous communication

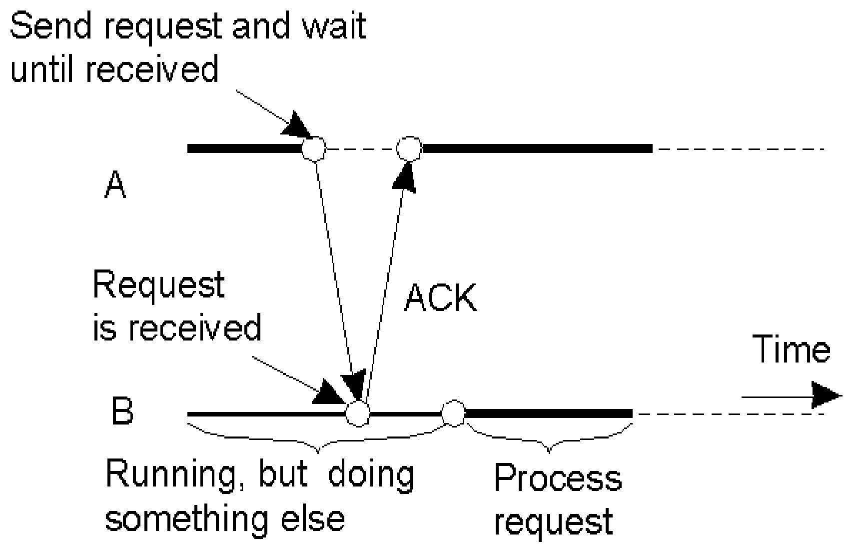
b) Persistent synchronous communication



Persistence & Synchronicity in Communications (/2)



(c)

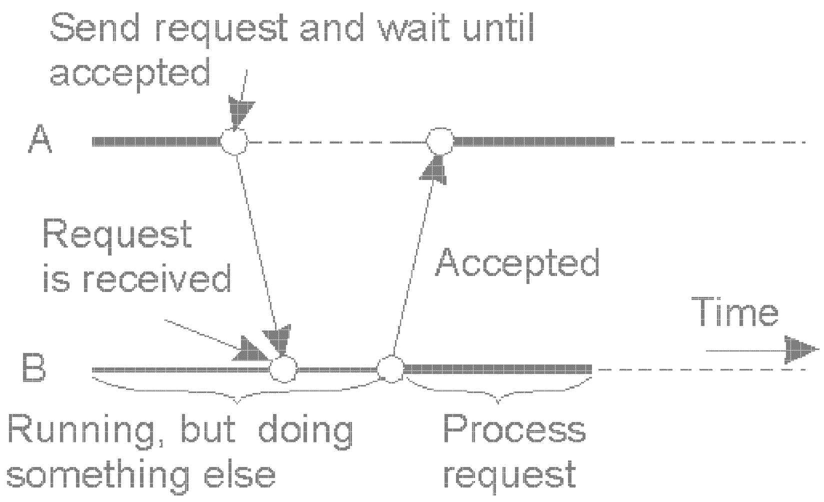


(d)

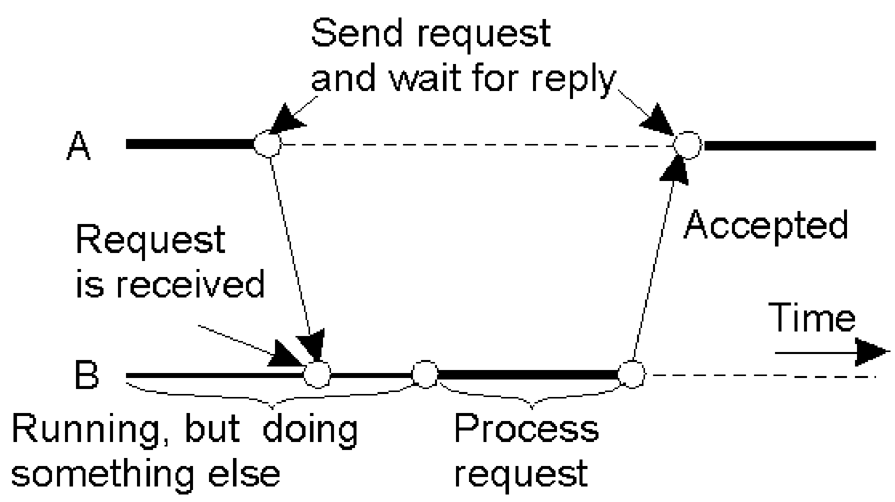
c) *Transient asynchronous communication*

d) *Receipt-based transient synchronous communication*

Persistence & Synchronicity in Communications (/3)



(e)



(f)

e) *Delivery-based transient synchronous communication at message delivery*

f) *Response-based transient synchronous communication*

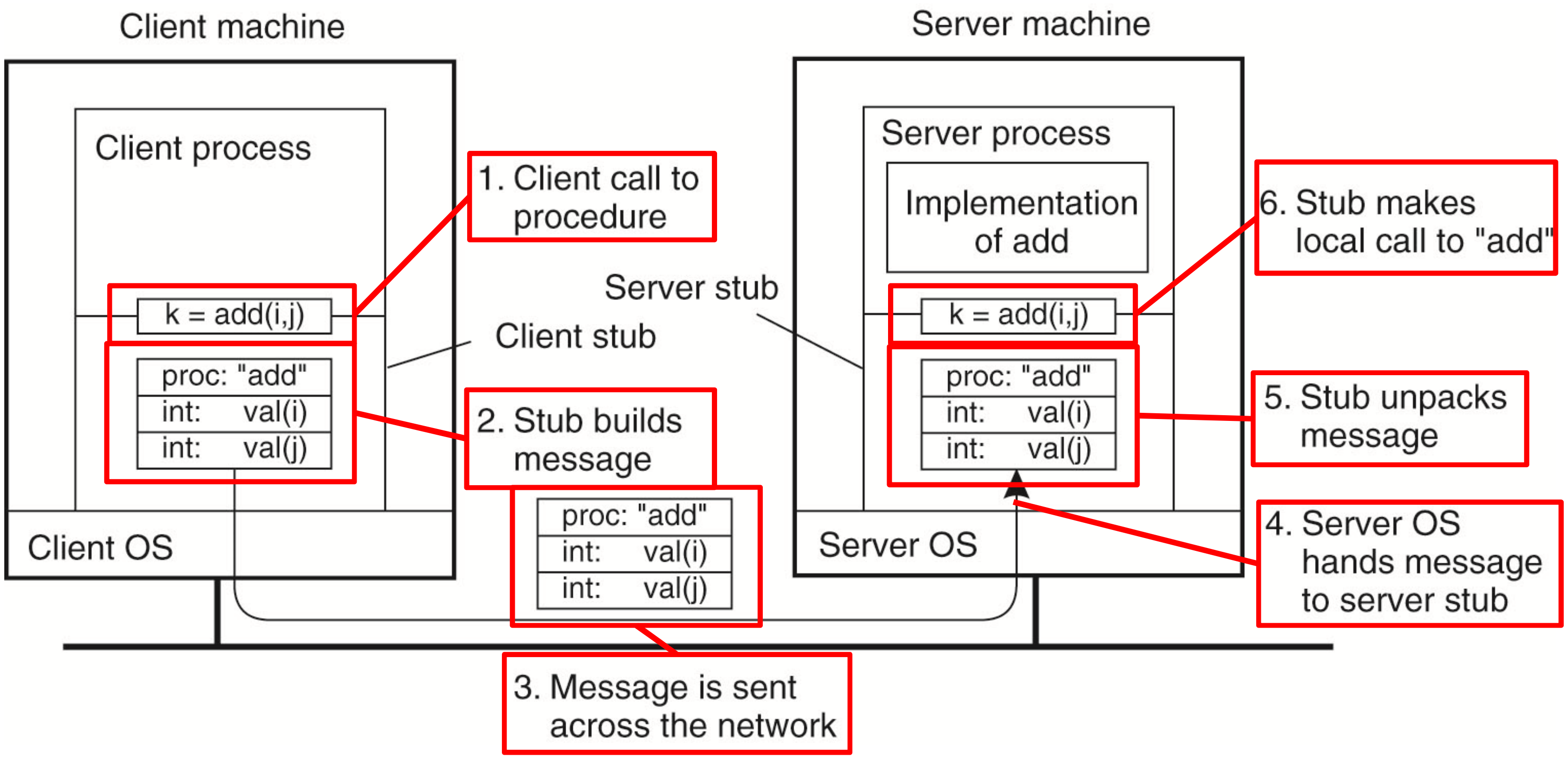
SECTION 6.3: REMOTE PROCEDURE CALL (RPC)

Remote Procedure Call (RPC)

- *Rationale:* *Why RPC?*
- *Distribution Transparency:*
  - Send/Receive don't conceal comms at all – need to achieve *access* transparency.
- *Answer: Totally New 'Communication' System:*
  - RPC allows programs to communicate by calling procedures on other machines.
- *Mechanism*
  - When a process on machine A calls a procedure on machine B, calling process on A is suspended,
  - Execution of the called procedure takes place on B.
  - Info 'sent' from caller to callee in parameters & comes back in result.
  - No message passing at all is visible to the programmer.
  - Application developers familiar with simple procedure model.

Basic RPC Operation

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



RPC: Parameter Passing

- *Parameter marshalling*

  More than just wrapping parameters into a message:

  - Client/server machines may have different data representations (e.g. byte ordering)
  - Wrapping parameter means converting value into byte sequence
  - Client and server have to agree on the same encoding:
    - How are basic data values represented (integers, floats, characters)?
    - How are complex data values represented (arrays, unions)?
  - Client and server need to properly interpret messages, transforming them into machine-dependent representations.

RPC: Parameter Passing (/2)

- *Assumptions Regarding RPC Parameter Passing:*
  - Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
  - All data to be operated on is passed by parameters. Excludes passing references to (global) data.

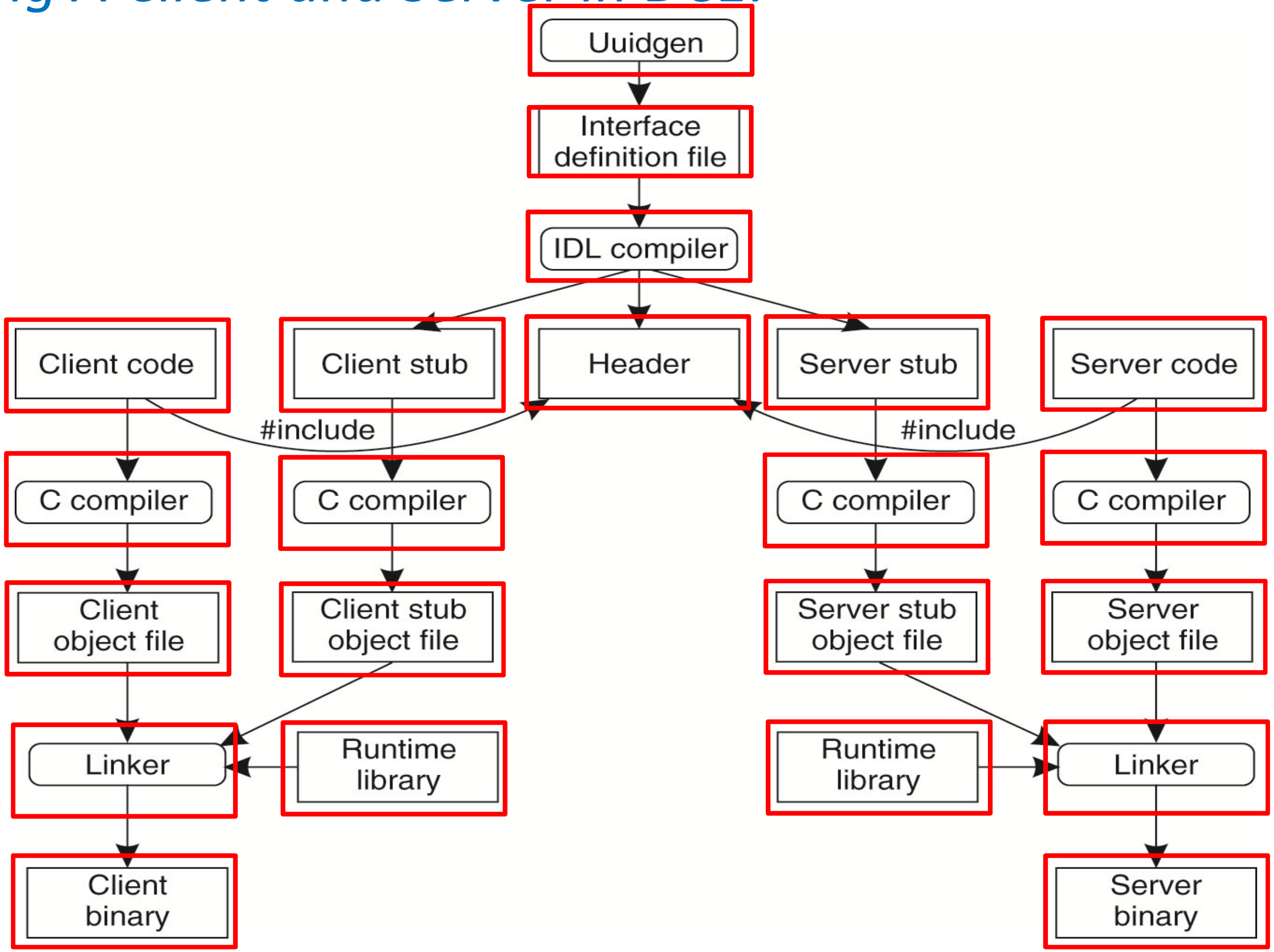
- *Conclusion*
  - Full access transparency cannot be realized

- *Observation:*
  - A remote reference mechanism enhances access transparency:

  Remote reference offers unified access to remote data
  - Remote references can be passed as parameter in RPCs

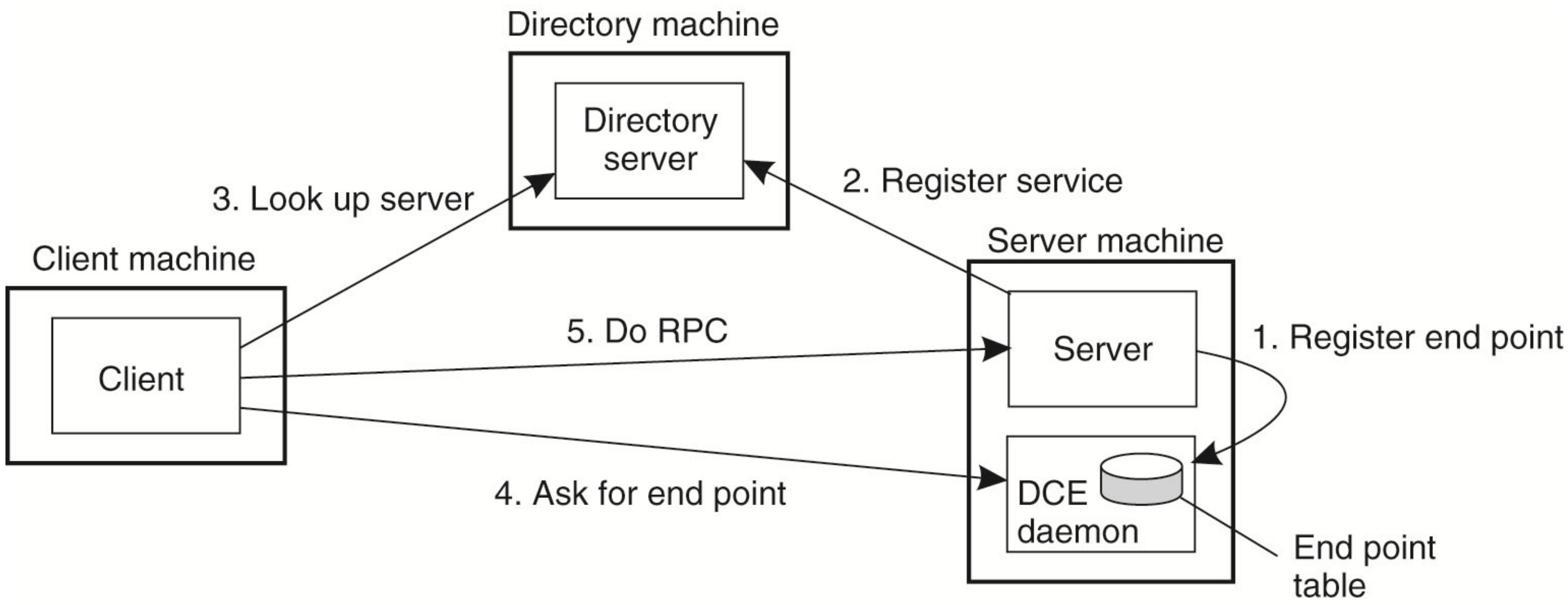
RPC Example: Distributed Computing Environment (DCE)

- *Writing A Client and Server in DCE:*



DCE Client to Server Binding

- *Registration & Discovery:*
  – Server registration enables client to locate server and bind to it.
  – Server location is done in two steps:
    1. Locate the server's machine.
    2. Locate the server on that machine.



SECTION 6.4: MESSAGE QUEUING SYSTEMS

Message-Oriented Persistent Comms

- *Rationale: Why Another Messaging System?:*
- *Scalability:*
  – "Transient" messaging systems, do not scale well geographically.
- *Granularity:*
  – MPI supports messaging O(ms). Distributed message transfer can take minutes
- *What about RPC?:*
  – In DS can't assume receiver is "awake" => default "synchronous, blocking" nature of RPC often too restrictive.
- *How about Sockets, then?:*
  – *Wrong level of abstraction (only "send" and "receive").*
  – *Too closely coupled to TCP/IP networks – not diverse enough*
- *Answer: Message Queueing Systems:*
  – MQS give extensive support for *Persistent Asynchronous Communication*.
  – Offer medium-term storage for messages – don't require sender/receiver to be active during message transmission.

Message-Oriented Persistent Comms. (/2)

- *Message Queuing Systems:*
  – *Basic idea*: applications communicate by putting messages into and taking messages out of “message queues”.
  – Only guarantee: your message will eventually make it into the receiver’s message queue => “loosely-coupled” communications.
  – Asynchronous persistent communication thro middleware-level queues.
  – Queues correspond to buffers at communication servers.
- *Four Commands:*

Primitive	Meaning
Put	Append a message to a specified queue.
Get	Block until the specified queue is nonempty, and remove the first message.
Poll	Check a specified queue for messages, and remove the first. Never block.
Notify	Install a handler to be called when a message is put into the specified queue.

Message-Queuing System Architecture

- *Operation:*
  - Messages are "put into" a *source queue*.
  - They are then "taken from" a *destination queue*.
  - Obviously, a mechanism has to exist to move a message from a source queue to a destination queue.
  - This is the role of the *Queue Manager*.
    - Function as message-queuing "relays" that interact with distributed applications & each other.
    - Not unlike routers, they support the idea of a DS "overlay network".

Role of Message Brokers

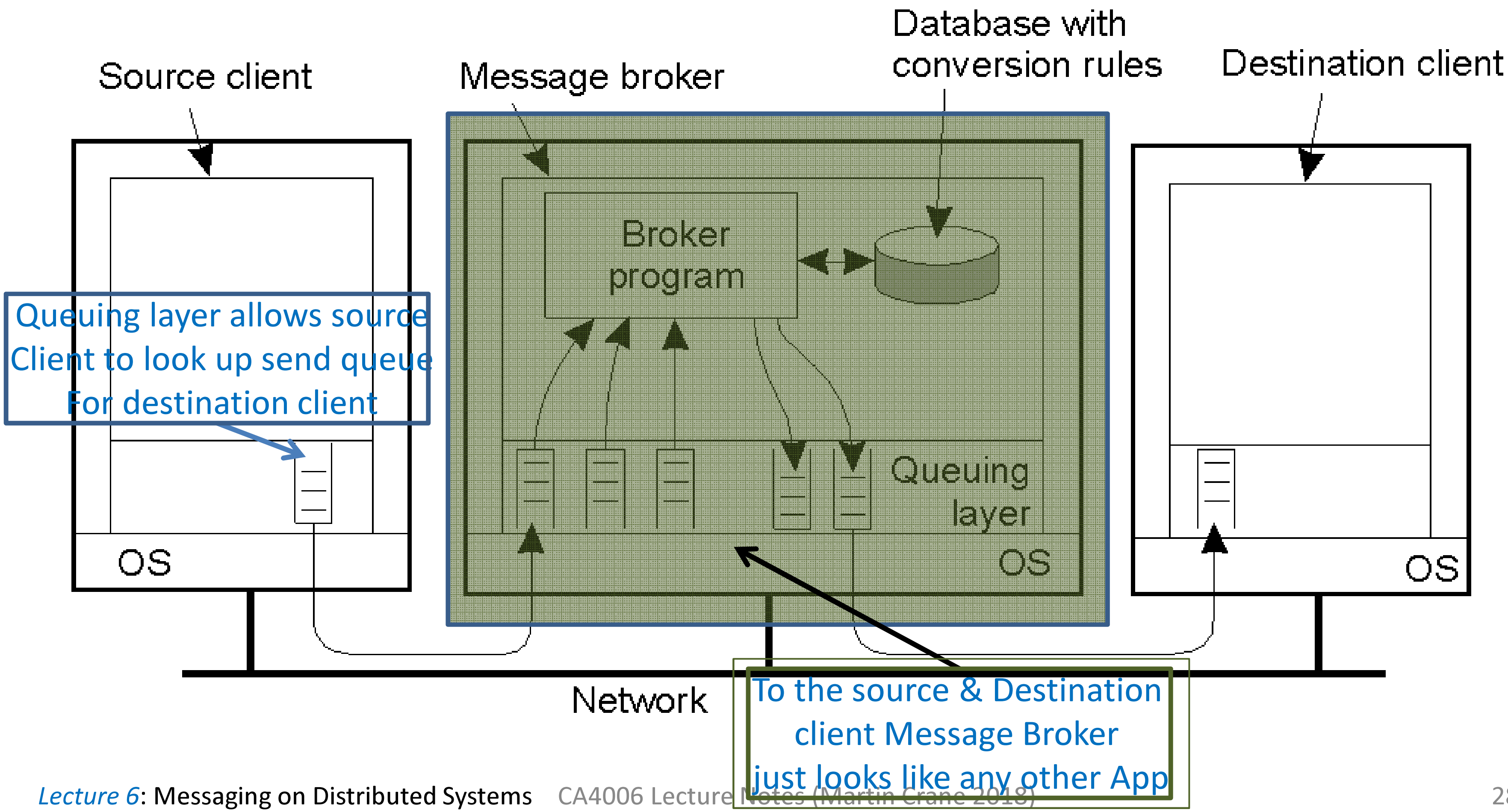
- *Rationale:*

Often need to integrate new/existing apps into a "single, coherent *Distributed Information System* (DIS)".

- *Problem:* different message formats exist in legacy systems
- Can't "force" legacy systems into single, global message format.
- "Message Broker" allows us to live with different formats
- Centralized component that takes care of application heterogeneity in an MQ system:
  - Transforms incoming messages to target format
  - Very often acts as an application gateway
  - May provide subject-based routing capabilities ⇒ *Enterprise Application Integration*

Message Broker Organization

- General organization of message broker in a MQS – also known variously as an “interface engine”.

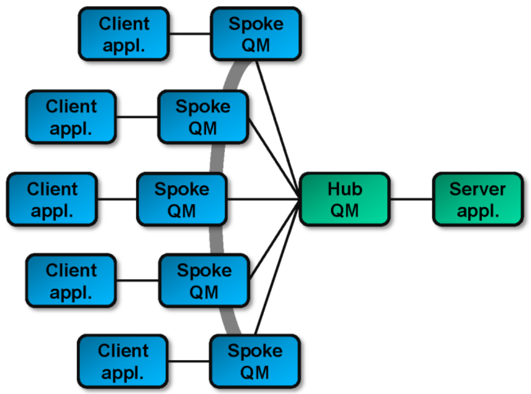


IBM's WebSphere MQ

- *Basic concepts:*
  - Application-specific messages are put into, removed from queues
  - Queues reside under the regime of a queue manager
  - Processes can put messages only in local queues, or thro an RPC
- *Message transfer*
  - Messages are transferred between queues
  - Message transfer btw process queues requires a channel
  - At each endpoint of channel is a *message channel agent*
  - Message channel agents are responsible for:
    - Setting up channels using lower-level n/w comm facilities (e.g. TCP/IP)
    - (Un)wrapping messages from/in transport-level packets
    - Sending/receiving packets

IBM's WebSphere MQ (/2)

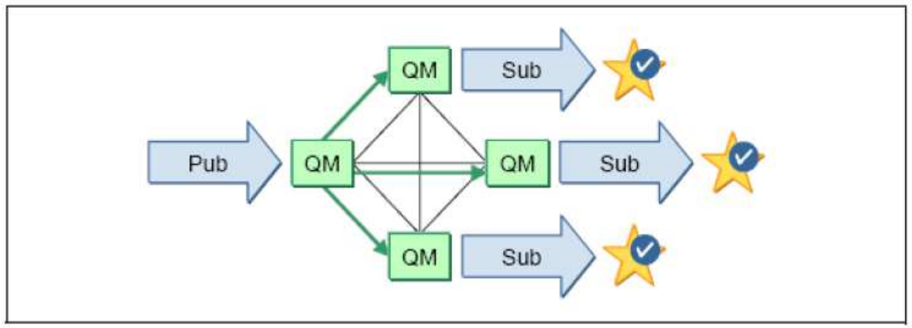
- Supported Topologies are:
  1. *Hub/spoke* topology (point-to-point queues)
     - Apps subscribe to "their" QM.
     - Routes to hub QM def'd in spoke QMs.

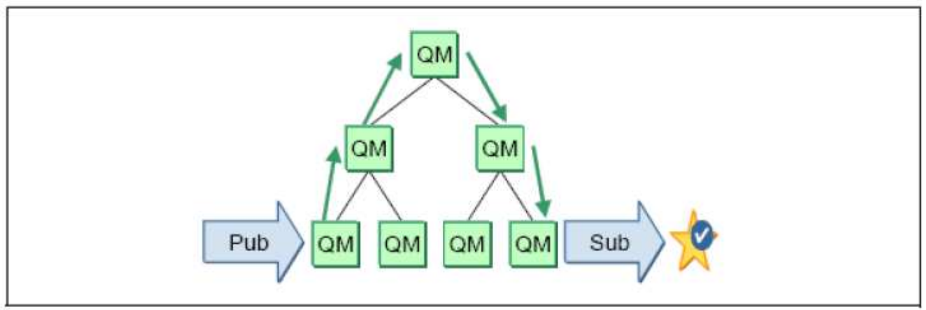


  2. *Distributed Publish/Subscribe*:
     - Apps subscribe to topics & publish messages to multiple receivers.
     - 2 Topologies: *Clusters* and *Trees*:

       Cluster: Cluster of QMs connected by channels. Published messages sent to all connected QMs of the published topic.

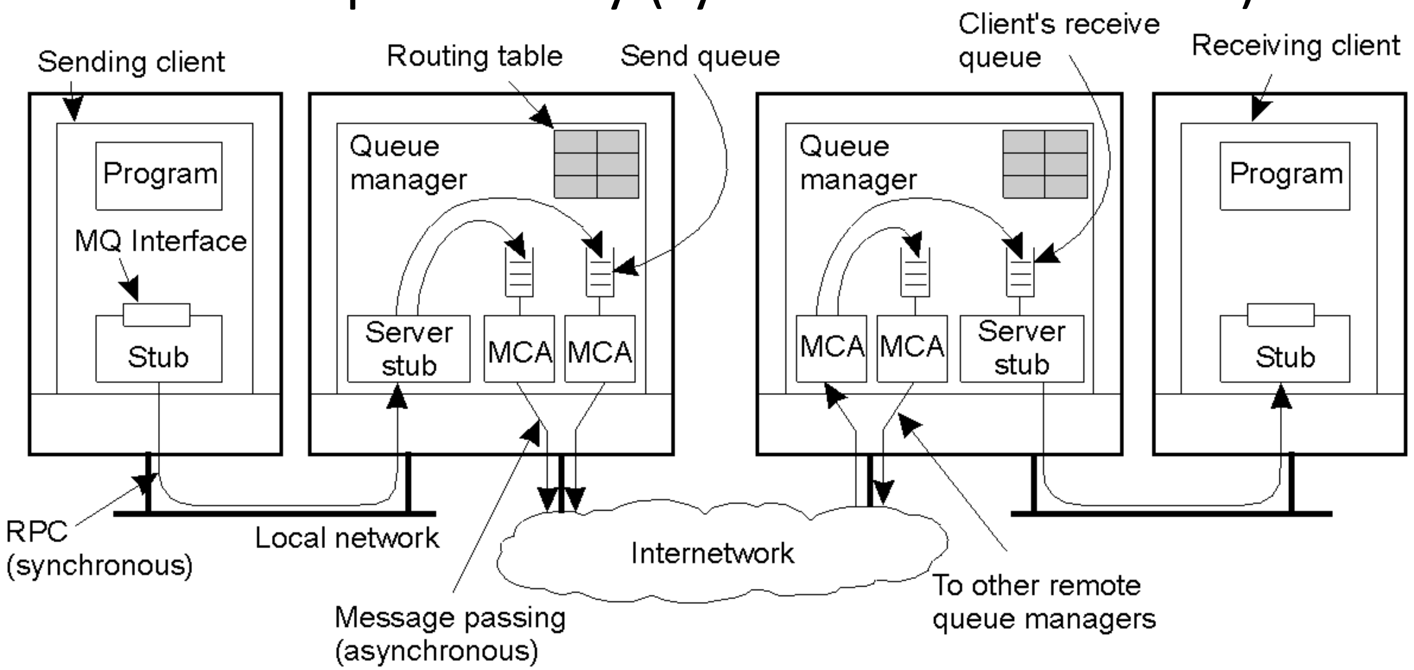
       Tree: Trees allow reducing number of channels between QMs.





IBM's WebSphere MQ (/2)

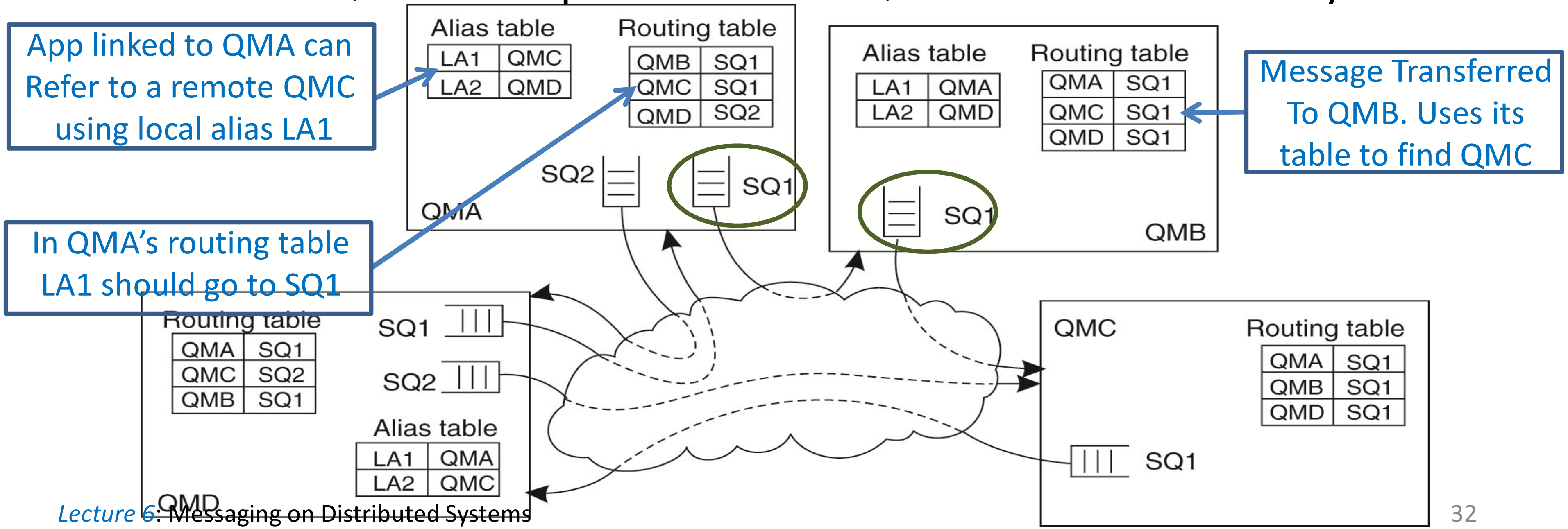
- *Principles of Operation:*
  – Channels are inherently unidirectional
  – Automatically start MCAs when messages arrive
  – Any network of queue managers can be created
  – Routes are set up manually (system administration)



General organization of IBM's WebSphere Message-Queuing System

IBM's WebSphere MQ (/3)

- *Routing:* Using logical names, in combination with name resolution to local queues, possible to route message to remote queue
  - Sending message from one QM to another (possibly remote) QM, each message needs destination address, so a transmission header is used
  - MQ Address has two parts:
    1. Part 1 is the *Destination QM Name* (say QMX)
    2. Part 2 is the *Name of the Destination Queue* (i.e. QMX's destination Queue)
  - As each QM has unique name each QM knows each other by an *Alias*



Advanced Message Queuing Protocol (AMQP)

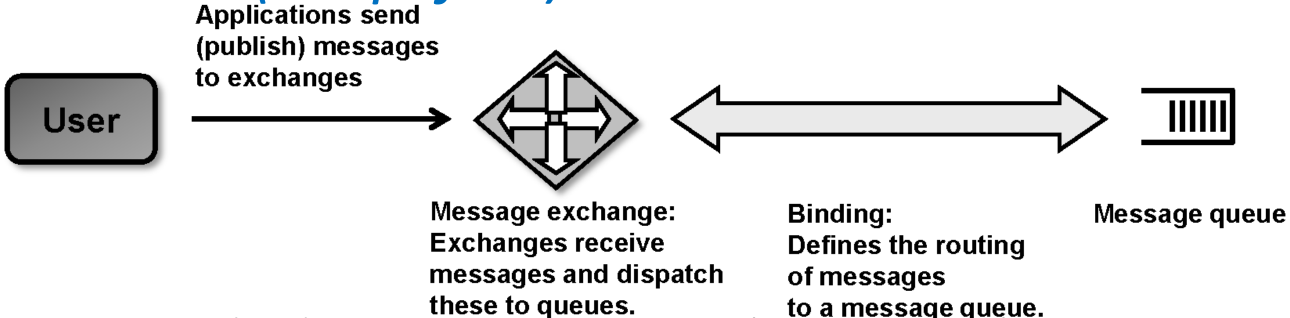
- *Why AMQP?*

1. Lack of standardization:
   - Little standardization in MOM products (mostly proprietary solutions).
     - E.g. 1: JMS Java- dependent, doesn't specify wire protocol only an API. => different JMS providers not directly interoperable on wire level.
     - E.g. 2: IBM Websphere clunky and expensive
2. Need for bridges[1] for interoperability:
   - To achieve interoperability between different queueing systems, 3rd party vendors offer *bridges*.
   - These complicate the architecture / topology, increase costs while reduce performance (additional delay).

[1]Entities that help in different stages of message mediation

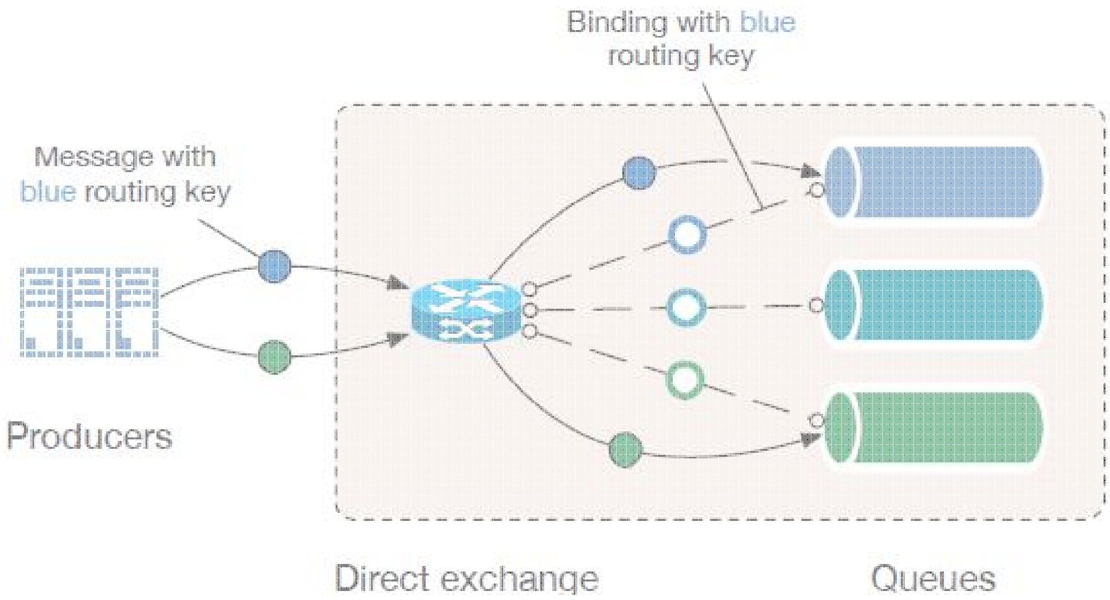
AMQP (/2)

- *Characteristics of AMPQ:*
  - What is it? Open protocol for enterprise messaging, supported by industry (JP Morgan, Cisco, Microsoft, Red Hat, Microsoft etc.).
  - Open/ Multi-platform / language messaging system.
  - AMQP defines:
    1. Messaging capabilities (called *AMQP model*)
    2. *Wire-level protocol* for interoperability
  - AMQP messaging patterns:
    1. Request-response: messages delivered to a specific queue
    2. Publish/Subscribe: messages delivered to a set of receiver queues
    3. Round-robin: message distribution to set of receivers based on availability
- *AMQP Model (simplified):*



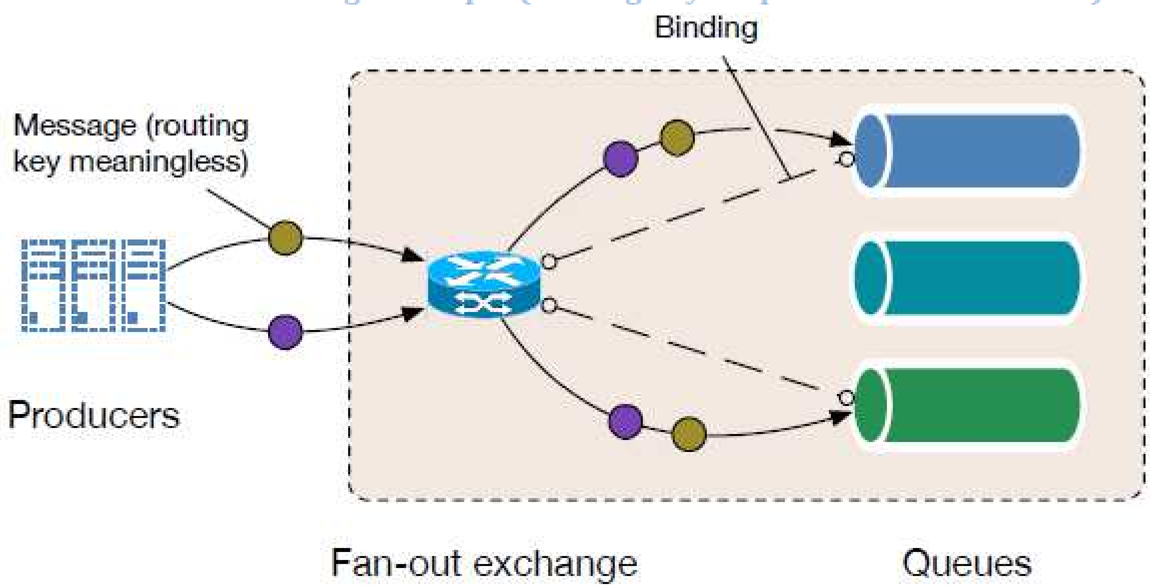
AMQP Example: RabbitMQ Model

Direct Exchange



- Direct exchange example (routing keys = queue names = colours)

Fan-out Exchange



- Fan-out exchange example (routing key meaningless, three queues, two bindings only)

Hello World in RabbitMQ

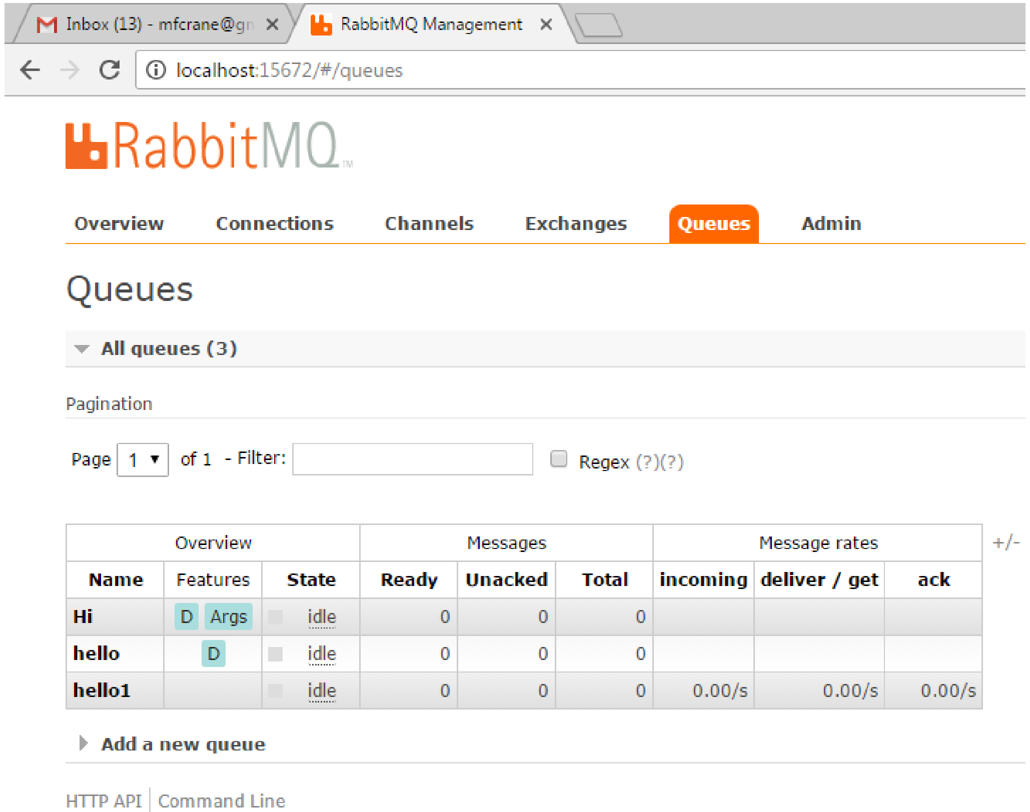
```
#!/usr/bin/env ruby
# encoding: utf-8
require "bunny"

conn = Bunny.new(:automatically_recover => false)
conn.start

ch   = conn.create_channel
q    = ch.queue("hello")  # create a message queue called “hello”

ch.default_exchange.publish("Hello World!", :routing_key => q.name)
# default_exchange is a direct exchange with no name
# main advantage is every queue is automatically bound to it with routing key same as queue name
puts " [x] Sent 'Hello World!'"

conn.close # close off the connection
```

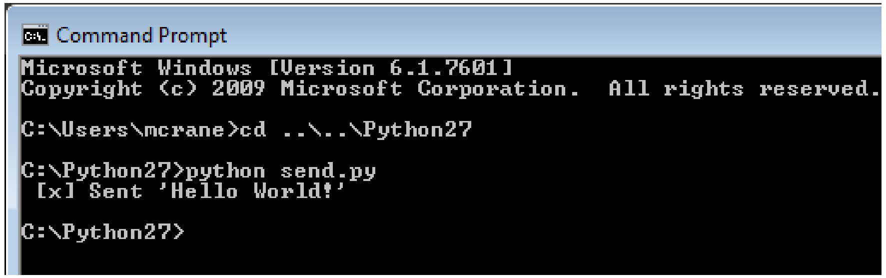
```
#!/usr/bin/env ruby
# encoding: utf-8
require "bunny"

conn = Bunny.new(:automatically_recover => false)
conn.start # if conn fails, reconnect tried every 5 secs, this disables automatic connection recovery

ch   = conn.create_channel
q    = ch.queue("hello") # create a message queue with same name as above

begin
 puts " [*] Waiting for messages. To exit press CTRL+C"
 q.subscribe(:block => true) do |delivery_info, properties, body|
   puts " [x] Received #{body}"
 end
rescue Interrupt => _ # exception handling if Interrupt happens (i.e. if CTRL+C hit)
 conn.close # close off the connection

 exit(0)
end
```

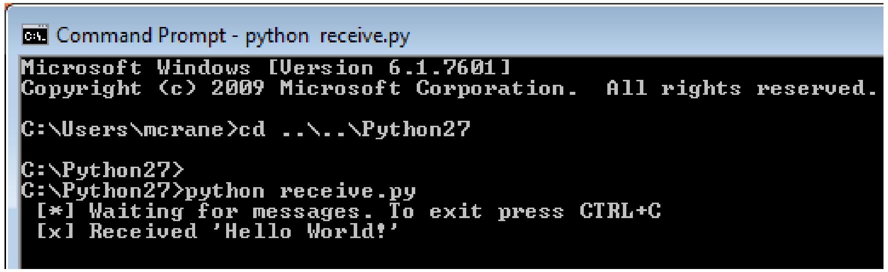
```
channel.basic_publish(exchange=‘ ‘,
                      routing_key='hello',
                      body='Hello World!')
```

RabbitMQ

- Afterwards should see something like this:







Work Queue In RabbitMQ

Command Prompt - python worker.py

```
C:\Users\mcrane>cd ..\..\Python27

C:\Python27>python worker.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Dasher'
 [x] Done
 [x] Received 'Vixen'
 [x] Done
 [x] Received 'Dunder'
 [x] Done
```

Command Prompt - python worker.py

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved

C:\Users\mcrane>cd ..\..\Python27

C:\Python27>python worker.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Dancer'
 [x] Done
 [x] Received 'Comet'
 [x] Done
 [x] Received 'Blixem'
 [x] Done
_
```

Command Prompt - python worker.py

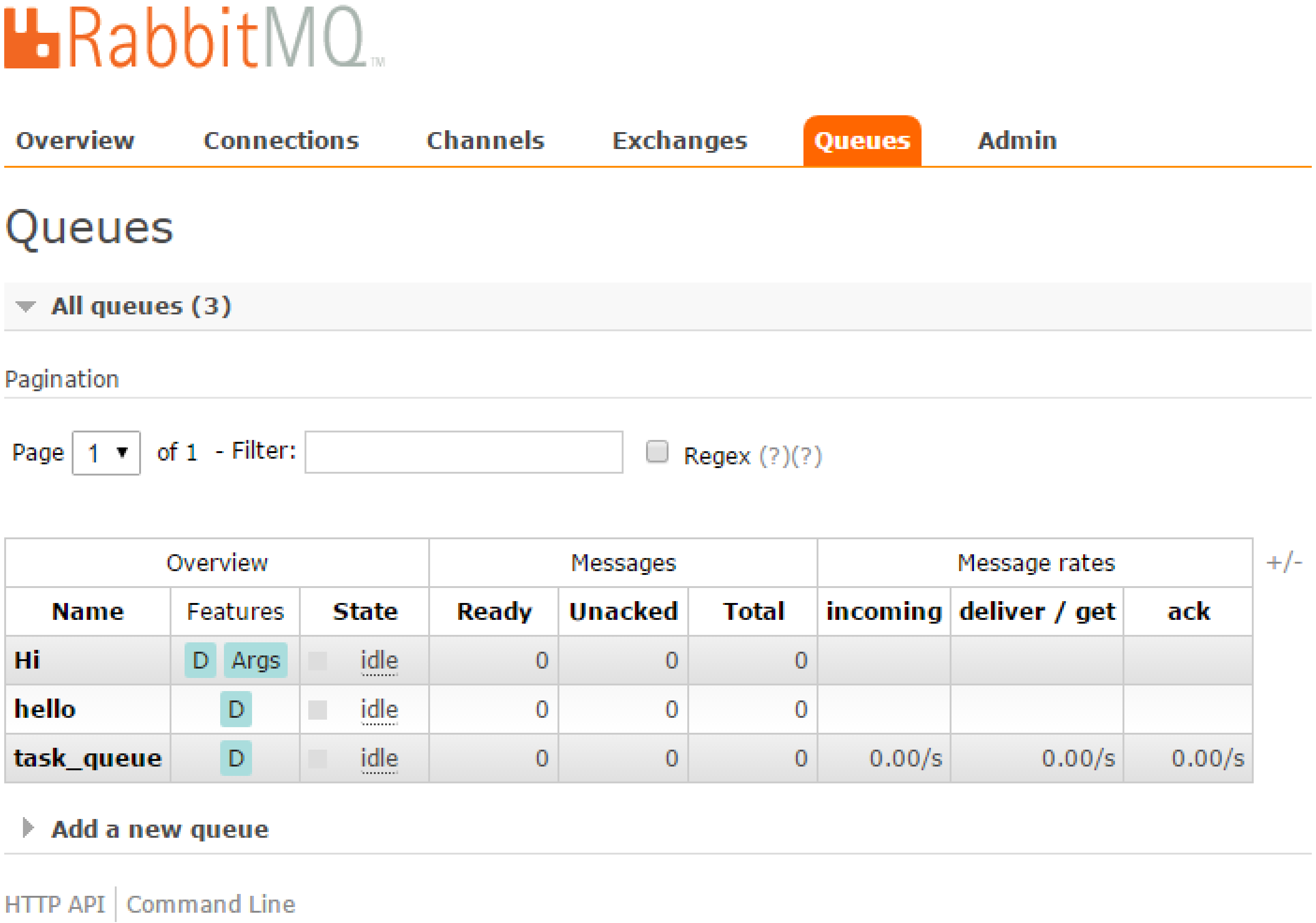
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserve

C:\Users\mcrane>cd ..\..\Python27

C:\Python27>python worker.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Prancer'
 [x] Done
 [x] Received 'Cupid'
 [x] Done
```

Command Prompt

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\mcrane>cd ..\..\Python27

C:\Python27>python new_task.py Dasher
 [x] Sent 'Dasher'

C:\Python27>python new_task.py Dancer
 [x] Sent 'Dancer'

C:\Python27>python new_task.py Prancer
 [x] Sent 'Prancer'

C:\Python27>python new_task.py Vixen
 [x] Sent 'Vixen'

C:\Python27>python new_task.py Comet
 [x] Sent 'Comet'

C:\Python27>python new_task.py Cupid
 [x] Sent 'Cupid'

C:\Python27>python new_task.py Dunder
 [x] Sent 'Dunder'

C:\Python27>python new_task.py Blixem
 [x] Sent 'Blixem'

C:\Python27>_
```

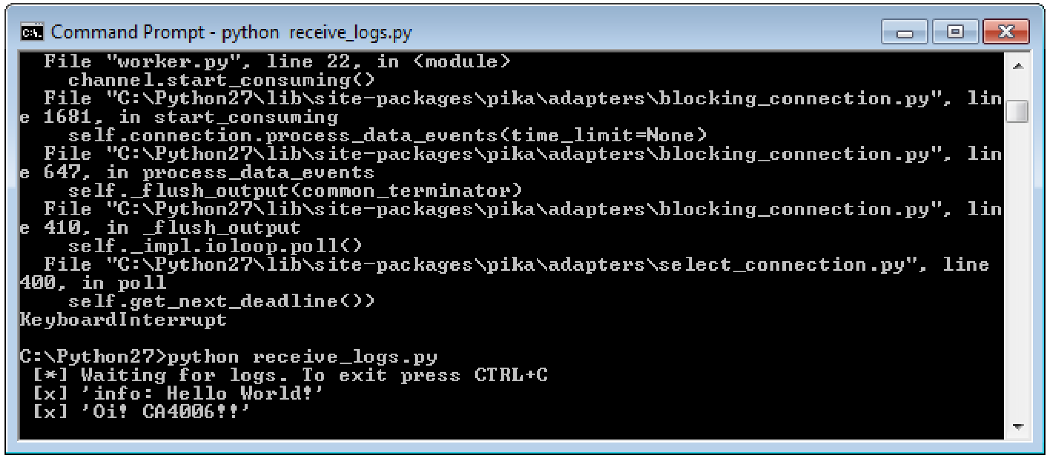
Work Queue In RabbitMQ (/2)

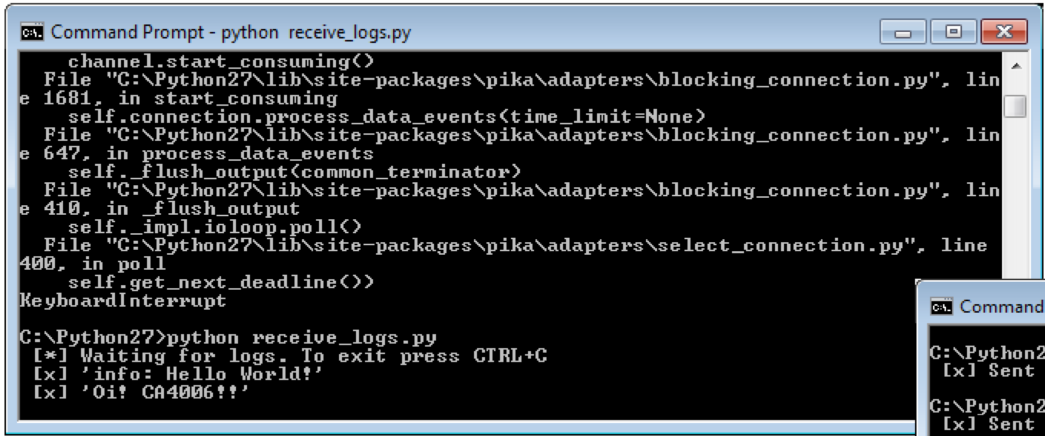
- Afterwards should see something like this:

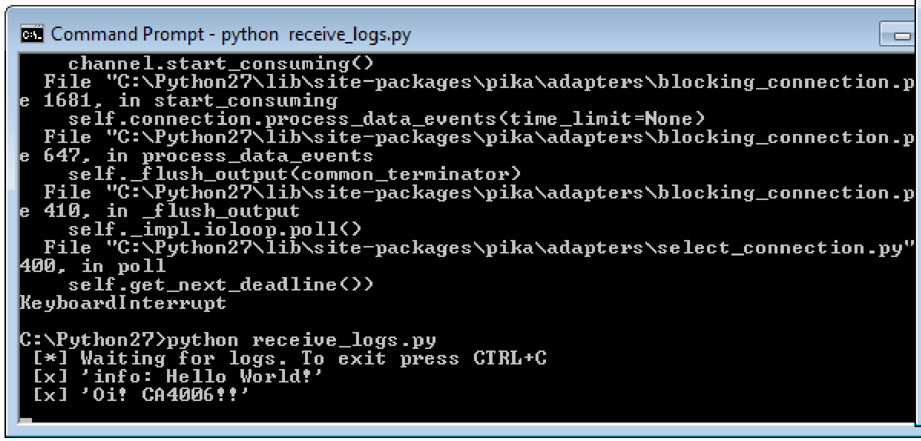


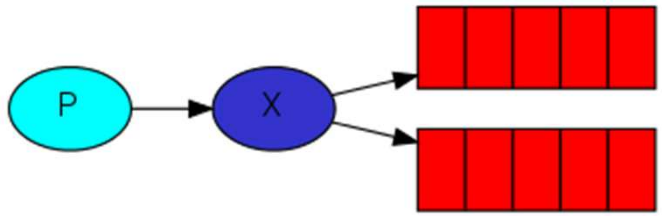
Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
Hi	D Args	idle	0	0	0			
hello	D	idle	0	0	0			
task_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

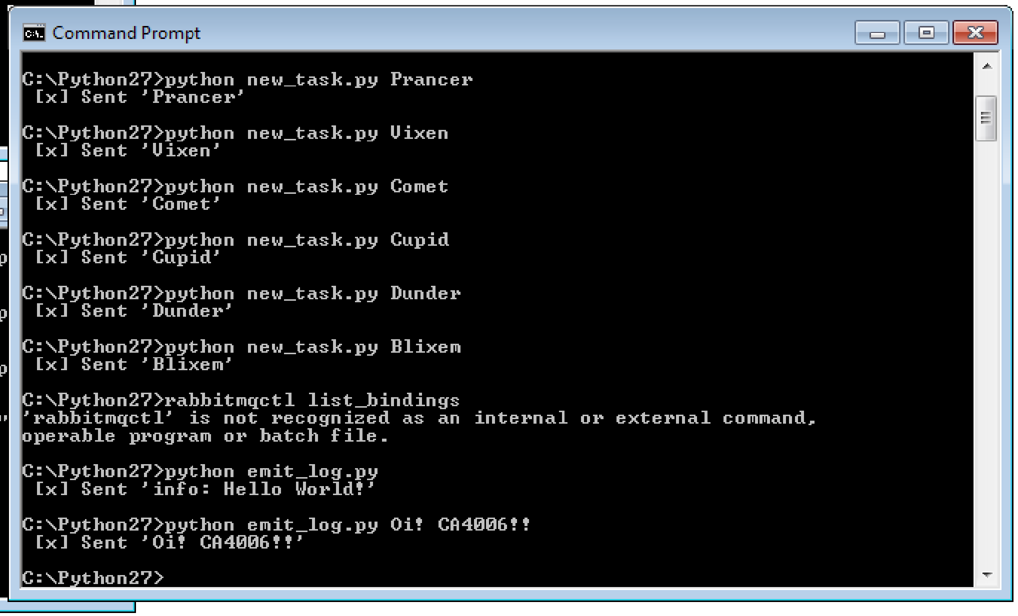
Publish-Subscribe In RabbitMQ



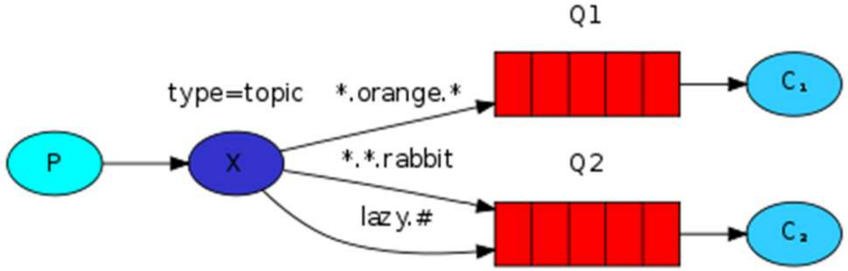








Topic-based Routing Publish-Subscribe In RabbitMQ



Command Prompt - python receive_logs_topic.py "CA4006.*" "CA4*"

```
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>python receive_logs_topic.py "CA4006.*" "CA4*"
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'CA4006.':'Assignment 2 is coming'
 [x] 'CA4*':'Time is running out on CA400'
```

Command Prompt - python receive_logs_topic.py "CA296" "CA2*"

```
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>python receive_logs_topic.py "CA296" "CA2*"
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'CA296':'CA296 is hard-study it!'
```

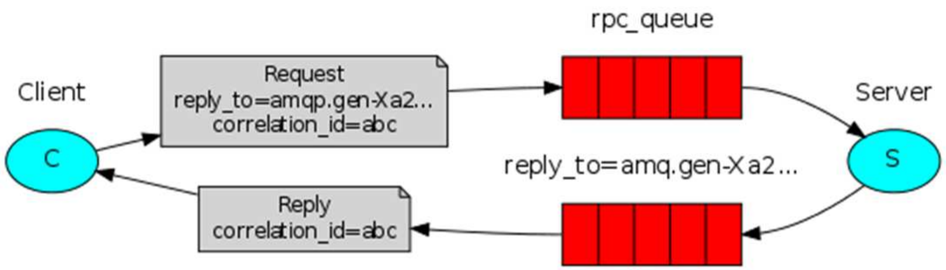
Command Prompt - python receive_logs_topic.py "CA4010.*" "CA4*"

```
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>python receive_logs_topic.py "CA4010.*" "CA4*"
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'CA4*':'Time is running out on CA400'
```

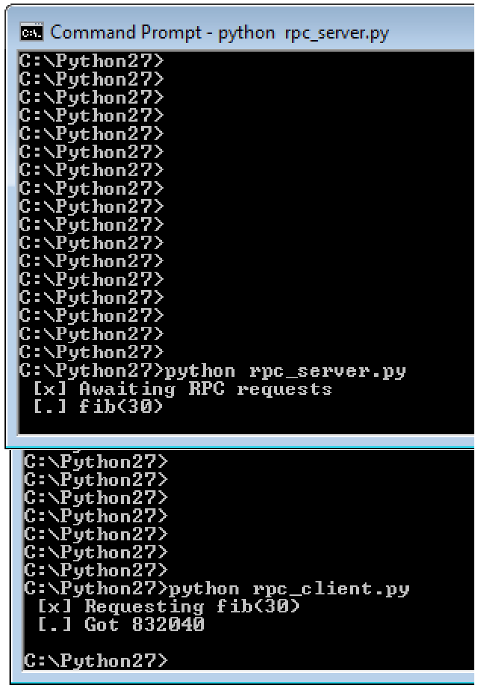
Command Prompt

```
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>python emit_log_topic.py "CA4006." "Assignment 2 is coming"
 [x] Sent 'CA4006.':'Assignment 2 is coming'

C:\Python27>python emit_log_topic.py "CA296" "CA296 is hard-study it!"
 [x] Sent 'CA296':'CA296 is hard-study it!'

C:\Python27>python emit_log_topic.py "CA40*" "Time is running out on CA400"
 [x] Sent 'CA40*':'Time is running out on CA400'

C:\Python27>python emit_log_topic.py "CA4*" "Time is running out on CA400"
 [x] Sent 'CA4*':'Time is running out on CA400'

C:\Python27>_
```

Fibonacci Server Using RPC In RabbitMQ



The RPC will work this:



- On startup, client creates anonymous exclusive `callback` Q
- For RPC request, Client sends a message with 2 properties: `reply_to` (set to the `callback` queue) & `correlation_id`, (a unique value for each request)
- The request is sent to an `rpc_queue` queue.
- RPC server awaits requests on that queue.
  - When a request comes, it does the job & returns a message with result to Client, using the queue from the `reply_to` field.
- Client awaits data on `callback` queue.
  - When one comes, it checks the `correlation_id` property.
  - If it matches the request's value it returns the response to the application.

SECTION 6.5: MULTICAST COMMUNICATION

Multicast Communication

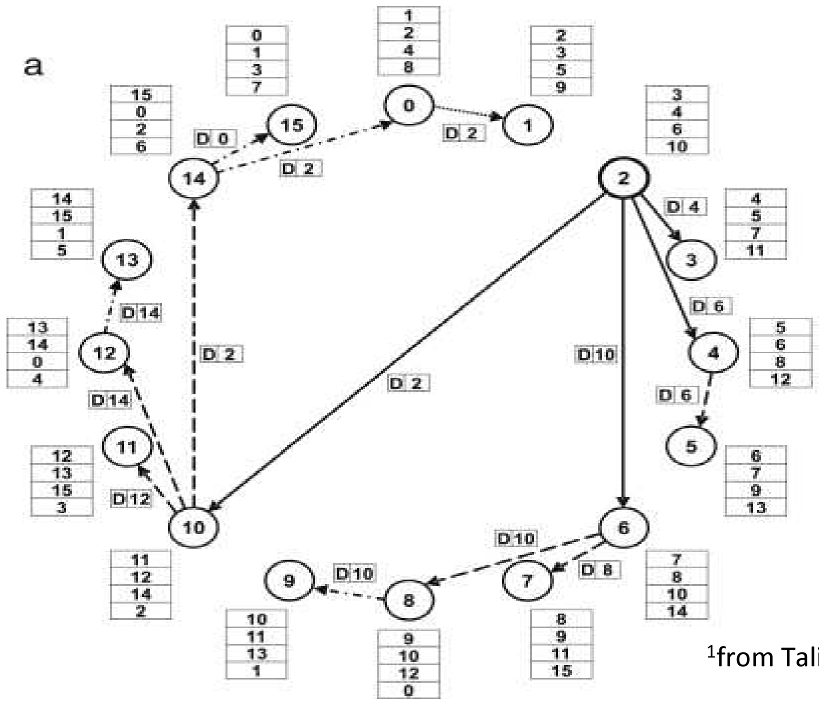
- *Rationale:* Often need to a *Send-to-Many* in Distributed Systems
- Examples:
  - *Financial services*: Delivery of news, stock quotes etc
  - *E-learning*: Streaming content to many students at different levels.
- *Problem:* IP Multicast is very efficient for bandwidth usage
- BUT key architectural decision: Add support for multicast in IP layer and no wide area IP multicast support
- *Solutions:*

1. *Application-Level Multicasting*
   - Nodes organize (e.g. with chord to build, maintain) into an overlay n/w,
   - Can then disseminate information to members
2. *Gossip-based data dissemination*
   - Rely on epidemic behaviour for data spreading

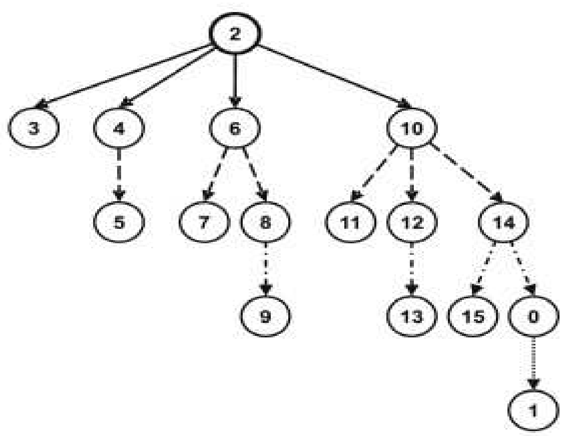
1. Application-Level Multicasting (ALM)

- *Basics:*
- In ALM, message sent over multicast tree created on overlay network
  - – Sender is the root of the tree which spans all the receivers
- A connection between two nodes may cross several physical links

  => ALM may incur more *cost* than network-level multicast (i.e. cross same physical link more than once)



b Multicast on Chord Network[1]



[1]from Talia & Trunfrio, J. Parallel & Dist Computing Vol(70(12)) pp1254 - 1265, 2010

2. Epidemic Algorithms

- *Essence:*
- Epidemic algorithms used to rapidly spread info in large P2P systems without setting up a multicast tree
- Assumptions:
  – All updates for specific data item are done at a single node (i.e., no write-write conflict)
  – Can distinguish old from new data as data is time stamped or versioned
- Operation:
  – Node receives an update, forwards it to randomly chosen peers (akin to spreading a contagious disease)
  – Eventually, each update should reach every node
  – Update propagation is lazy

2. Epidemic Algorithms (/2)

- *Glossary of Terms:*
  – Node is *infected* if it has an update & wants to send to others
  – Node is *susceptible* if it has not yet been updated/infected
  – Node is *removed* if it is not willing or able to spread its update or can no longer send to others for some reason.
- We study two propagation models here:
  – *Anti-entropy*

  Each replica regularly chooses another randomly & exchanges state differences, giving identical states at both afterwards.

  – *Gossiping:*

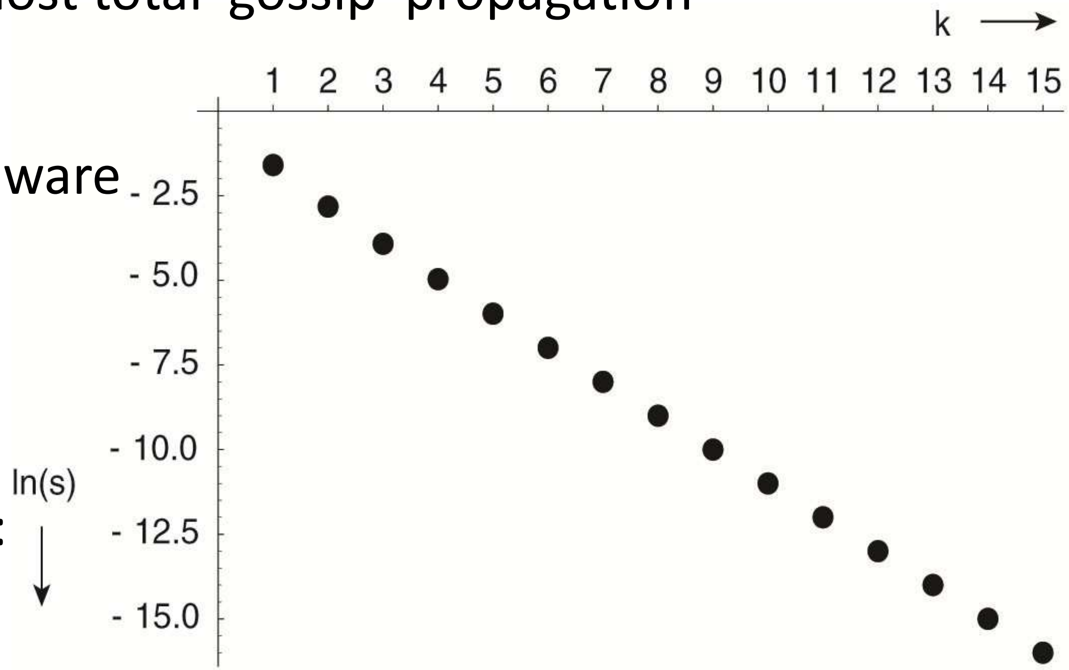
  A replica which has just been updated (i.e., has been infected), tells other replicas about its update (infecting them as well).

2. Epidemic Algorithms (/3)

- *Principal Operations of Anti-Entropy:*
  - A node P selects another node Q from the system at random.
    - *Push*: P only sends its updates to Q
    - *Pull*: P only retrieves updates from Q
    - *Push-Pull:* P and Q exchange mutual updates (after which they hold the same information).
- *Observations*
  - For push-pull it takes $O(\log(N))$ rounds to disseminate updates to all N nodes (*round*= when every node has initiated an exchange).
  - Anti-Entropy is reliable but costly (each replica must regularly choose another randomly)

2. Epidemic Algorithms (/4)

- *Basic model of Gossiping:*
  - A server S having an update to report, contacts other servers.
  - If a server is contacted to which update has already propagated, S stops contacting other servers with probability $^{1}/_{k}$.
  - i.e. increasing k ensures almost total 'gossip' propagation
- *Observations*
  - If s is fraction of servers unaware of update, can show that with many servers, the equation $s = e^{-(k+1)(1-s)}$ is satisfied
  - Example: for 10,000 servers: when $k = 4, s < 0.007$



  - If need 100% propagation, gossiping alone is not enough, maybe need to run one round of anti-entropy.

2. Epidemic Algorithms (/5)

- *The Deletion Problem in Epidemic Algorithms:*
  - Cannot remove old value from a server, expecting removal to propagate.
  - Instead, mere removal will be undone in time using epidemic algorithms
- *Solution:* Must register removal as special update by inserting a death cert
- *Next problem:*
  - When to remove a death certificate (it is not allowed to stay for ever)?
    - Run a global algorithm to detect if removal is known everywhere, and then collect the death certificates (looks like *garbage collection*) or
    - Assume death certificates propagate in finite time, and associate max lifetime for a certificate (can be done at risk of not reaching all servers)
  - Note: It is necessary that a removal actually reaches all servers.
- Applications of Epidemic Algorithms:
  - (Obviously) data dissemination
  - Data aggregation: each node with value x_i. Two nodes gossiping should reset their variable to $(x_i + x_j)/2$. What final value will nodes possess?

Lecture Summary

- Middleware enables much functionality in DS
- Especially the many types of interaction/communications necessary
- With rational reasons for every one!
  - *Remote Procedure Call* (RPC) enables transparency
  - But *Message Queuing Systems* necessary for persistent communications
    - IBM Websphere is ok but a bit old, clunky & tired at this stage?
    - AMQP open source, more flexible, better Industrial support?
  - *Multicast Communications* are often necessary in DS:
    - Application Layer Messaging (ALM)
    - Epidemic Protocols