LECTURE 3

Communications

Why communications?

- When computers need to exchange information to perform a computation (joint), they need to communicate.
- In this slide set, we will study two types of communications:
  - Remote Procedure Calls (RPC)
  - Message Oriented Communications

Chapter 4 of book

What are RPCs?

- Distributed systems are especially suited for providing services (e.g., MapReduce)
- Explicit message passing can be used for communications between processes – but do not conceal communications.
- How do we access services ?
  - Simply allow programs to call procedures that are located execute on other machines
    - Process 1 on Machine A calls Procedure2 on Machine B.
    - Blocks and waits for the procedure to return
    - Continues execution
  - Message passing hidden from programmer
- This method is called Remote Procedure Call or RPC for short
- RPC is widely used and forms the basis for communications in many distributed systems.

Challenges

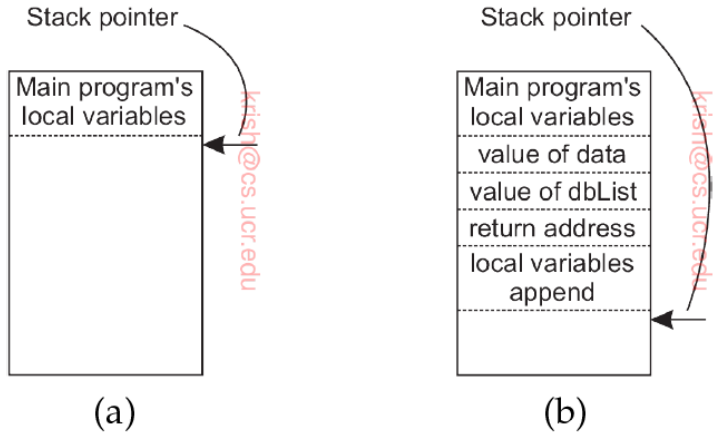
- While the basic idea is simple, hard to implement.
- Procedure executes on a different machine with a different address space.
- Passing parameters and results is necessary.
- Fault tolerance and synchronization

Arguments

- Conventionally, arguments are passed by value or by reference (pointers).

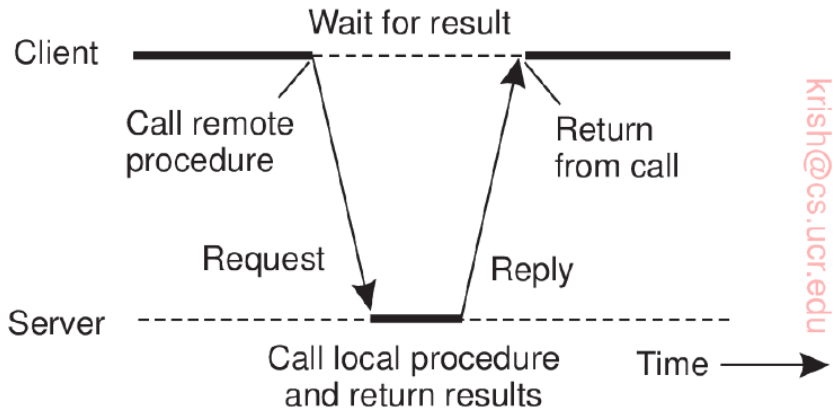
  e.g., append(data, dBList);

  - Copy-by-value or Copy-by-reference
  - The data may be values of local variables and dBList maybe a pointer to a list.
  - These are then pushed to the Stack
  - Programmer doesn't need to Know what append does. Only pass arguments.



Principles of RPC

- The procedure is executed on a remote machine.
  - What do we do ?
- There is a different version of append() running on the client.
  - Called a client stub
- It does not perform an append operation – instead it packs the parameters into a message which is sent to a server.
- Using send()



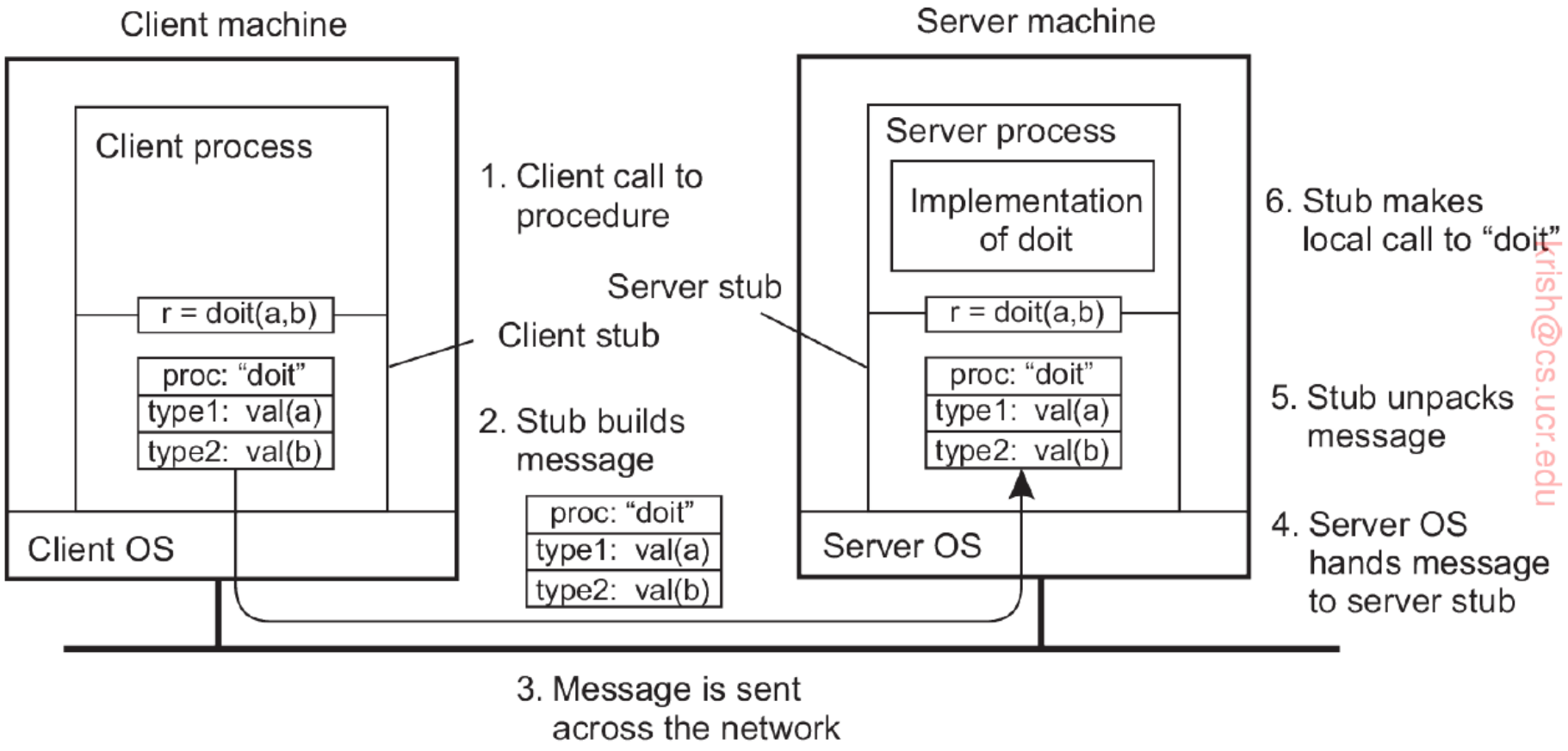
Principle of RPC (2)

- The client then blocks itself until the reply from server gets back.
- Server's OS passes message to a **server stub**.
  - Equivalent to client stub.
  - Waits for incoming requests by calling receive()
- Unpacks parameters from message and then calls the server procedure in the usual way.
  - As if being called directly by the client.
- Performs its work and returns result.

Principles of RPC (3)

- When the result is received, (using receive()) it unblocks the client stub.
- The client stub inspects message, unpacks result, and copies it to the caller.
  - It returns in the usual way.
  - The caller gets control – and all it knows is that the data has been appended to dBList.

Steps of RPC



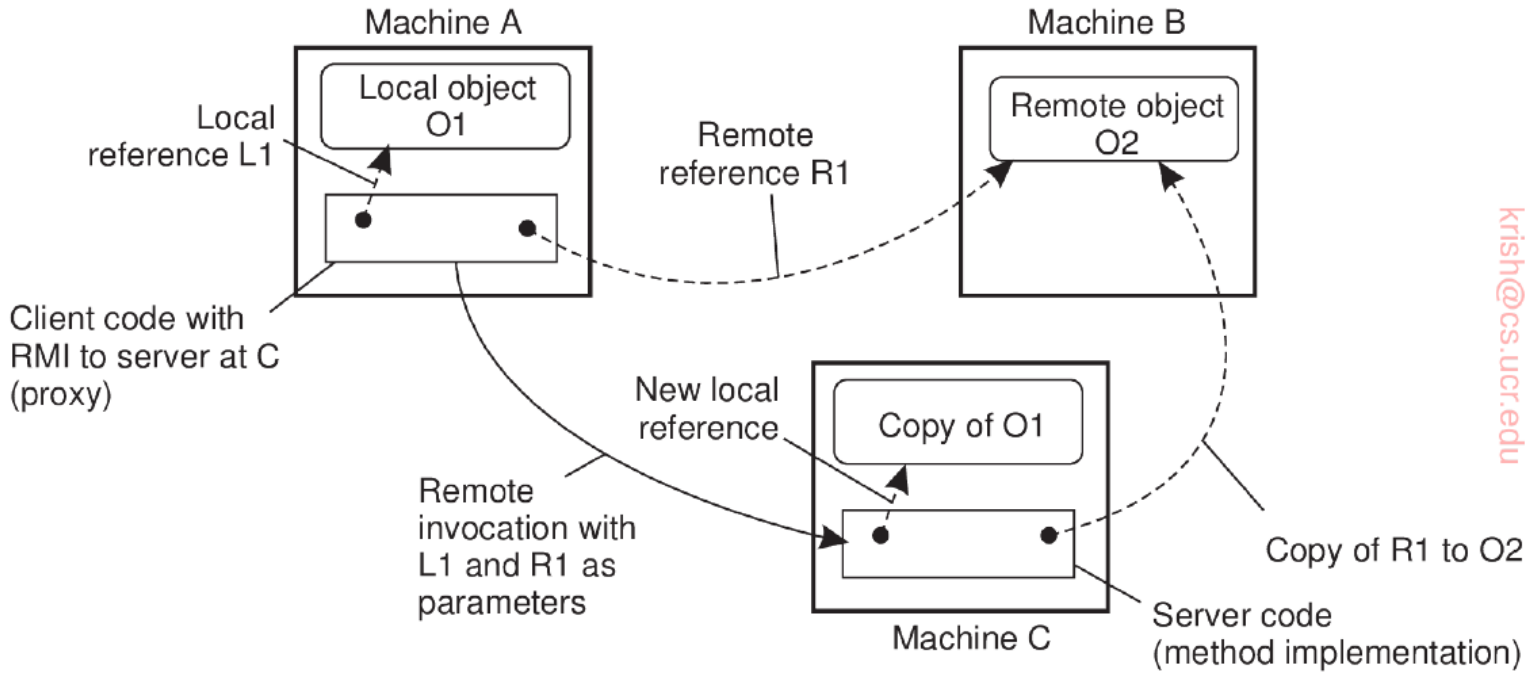
Parameter marshaling

- Packing parameters into messages is called parameter marshaling.
- Needed because the parameters (e.g., data and dBList) sent over the network are correctly interpreted.
- Little Endian vs Big Endian byte ordering
  - Standardized to Big Endian
- Transform the data to a machine and network independent format (essential)
  - Can be done with programming support.

Pointers

- How are pointers or references passed?
  - Cannot just forbid pointer and reference parameter passing.
- Solution : Copy the entire data structure
  - Effectively replaces copy-by-reference to copy-by-value/restore (return restores the reference)
  - Not semantically exact but good enough
- When client knows that the referred data is read only – no need for restore.

More complex parameter passing



The Machine A is executing a RPC on Machine C; There is a local object and a remote object.

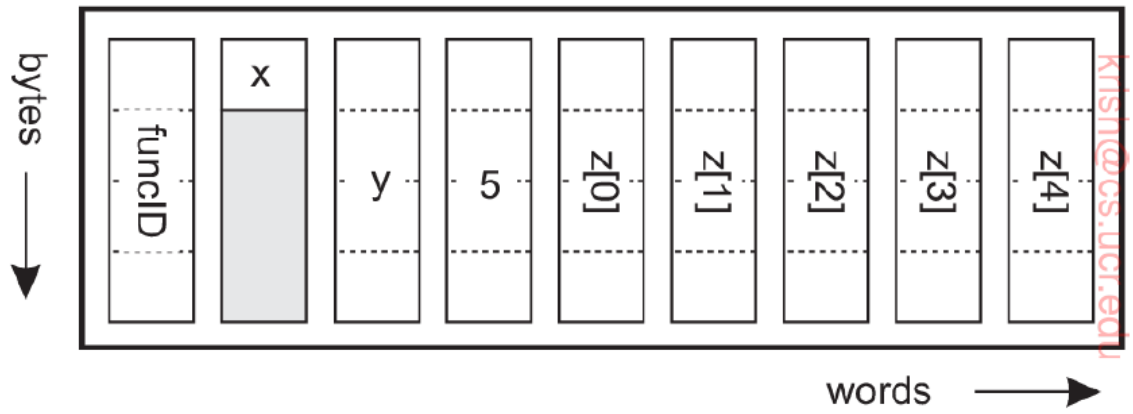
Interface Definition Language

- Often specified by means of an Interface Definition Language (IDL).
  - Example the DCE RPC – from the Open Software Foundation
- Defining the message format
- Representation of different (simple) data structures such as integers, characters etc.
- Protocol of use (e.g., TCP)

Stub Generation

```
void someFunction(char x; float y; int z[5])
```

(a)

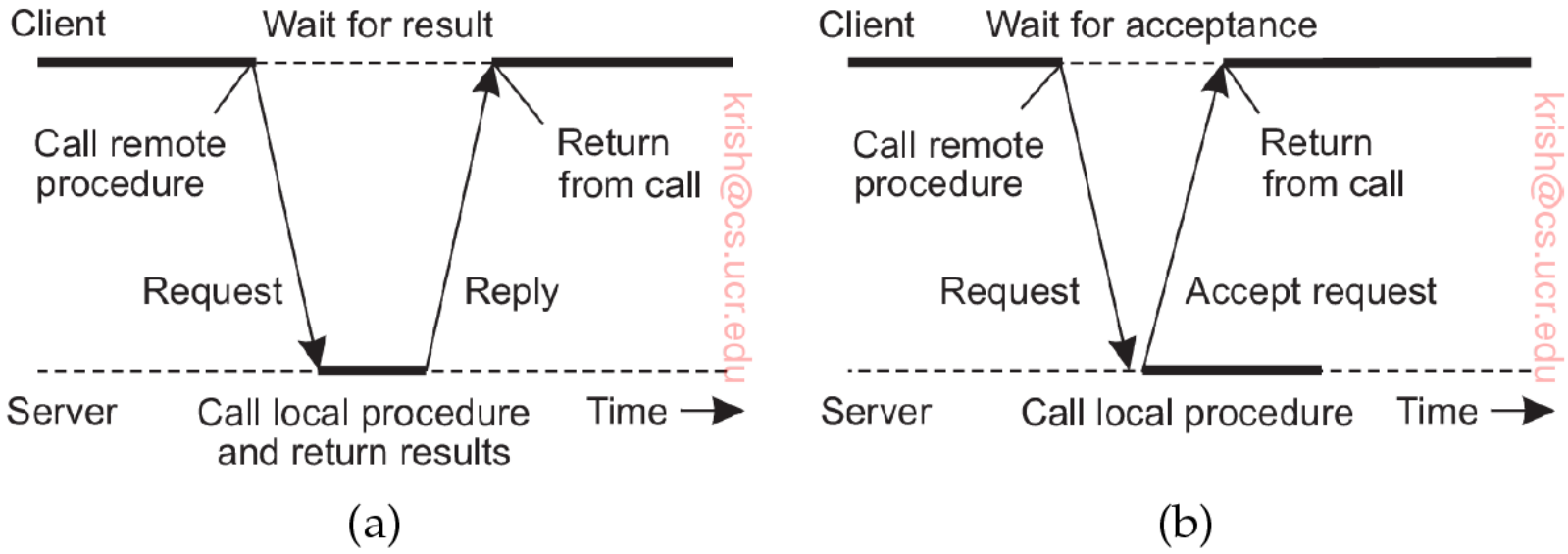


(b)

- Rules for how to encode.
- Example: Character in rightmost byte of a word, with the following 3 bytes empty.
- A float is a whole word
- An array is a group of words equal to the array length; preceded by size.

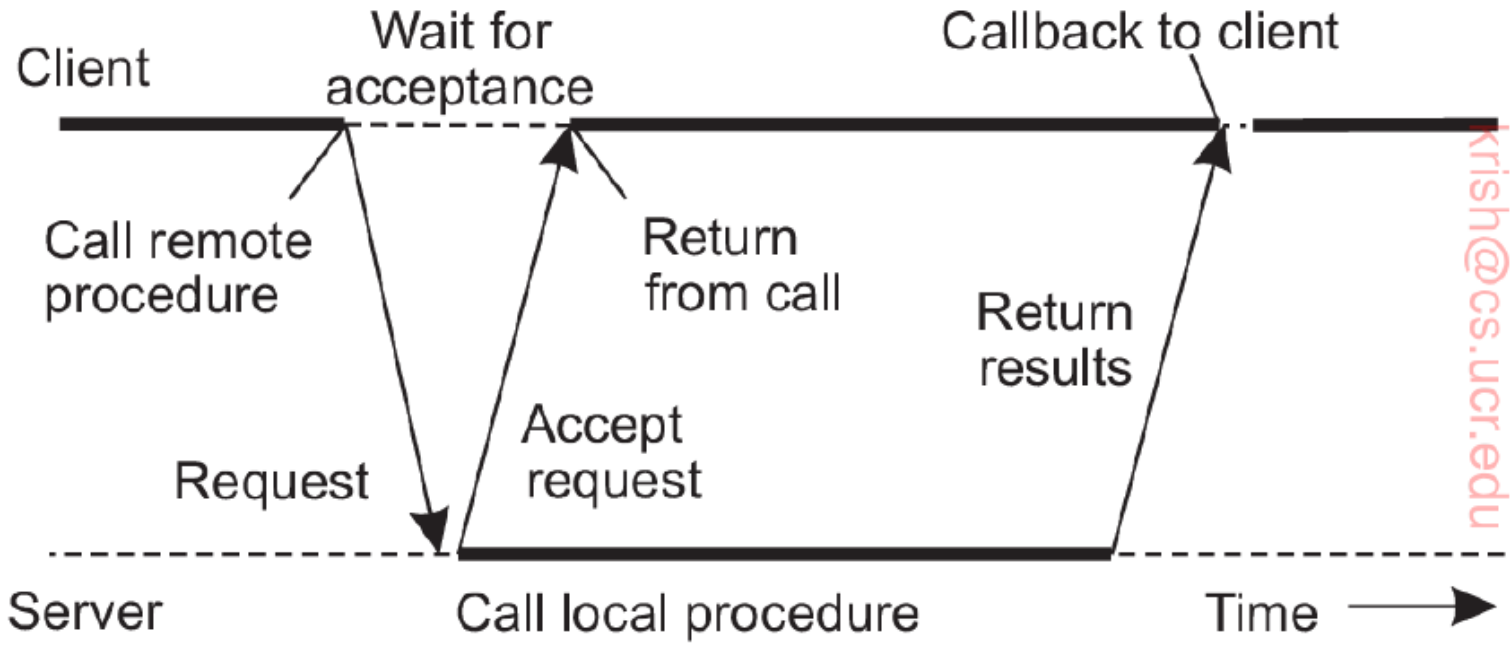
Asynchronous RPC

- Sometimes the RPC does not have to return a result to the client
- Blocking an issue
- Asynchronous RPC eliminates this issue.
  - Server ACKS request at which point client continues.



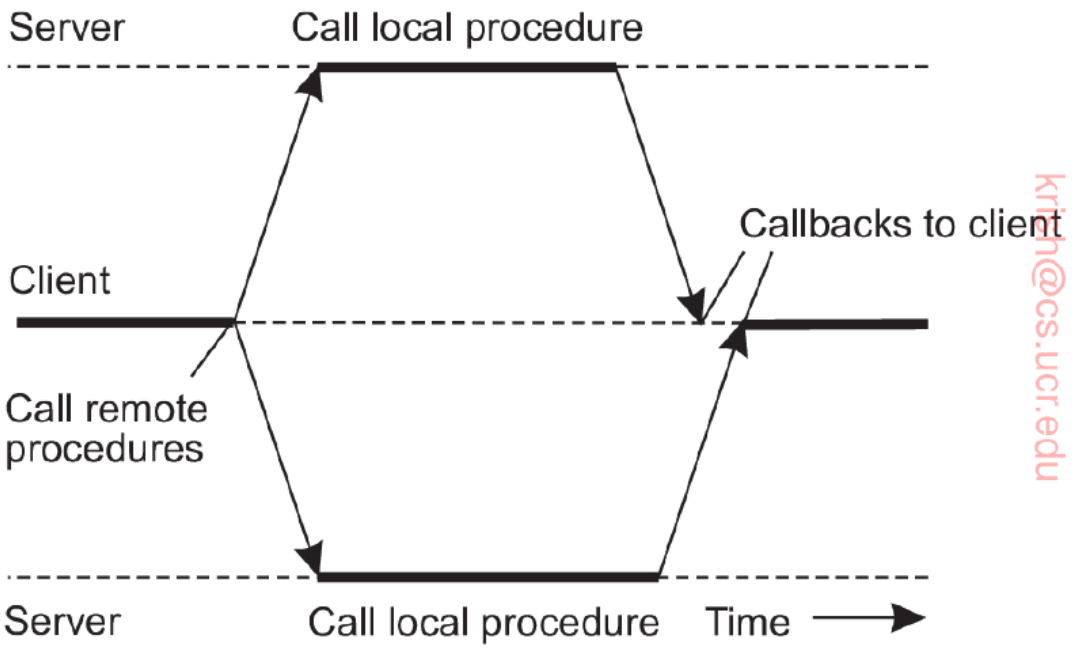
Asynchronous RPCs with returns

- Client may not want to wait for the RPC to return
  - For example, MapReduce where client interacts with many servers simultaneously.
- Combine RPC with **callback** (a user defined function which is invoked upon a certain event like message receipt)



Multicast RPC

- Allows sending procedure requests to multiple servers (e.g., MapReduce)



Message-oriented communication

- RPC enhance access transparency (not evident to user).
- Not sure that the receiving side is executing at the time a request is issued.
- Messaging solves this problem

Sockets

- Many distributed systems and applications leverage the underlying transport protocol (TCP).
  - Offers reliability
- Uses socket interface
- A communication interface that is exported to the user space via APIs.
- Applications can write data that are to be sent over the underlying network.

Socket operations in TCP/IP

Operation	Description
socket	Create a new communication end point
bind	Attach a local address to a socket
listen	Tell operating system what the maximum number of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some data over the connection
receive	Receive some data over the connection
close	Release the connection

- Socket operations need to be executed both on the client and server sides.

Server side operations

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
  - The server forks off a process which handles the actual connection.
- Send and receive data

Client side operations

- Create a socket with the socket() system call
- Connect the socket to the address of the server using the connect() system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.
  - Simply read or write to the socket.

Example: server

```
1  from socket  import *
2  s = socket(AF_INET, SOCK_STREAM)
3  s.bind((HOST, PORT))
4  s.listen(1)
5  (conn, addr) = s.accept()  # returns new socket and addr. client
6  while True:                # forever
7    data = conn.recv(1024)   # receive data from client
8    if not data: break       # stop if client stopped
9    conn.send(str(data)+"*") # return sent data plus an "*"
10 conn.close()               # close the connection
```

Figure 4.20: (a) A simple socket-based client-server system: the server.

- Server adds an * and returns data

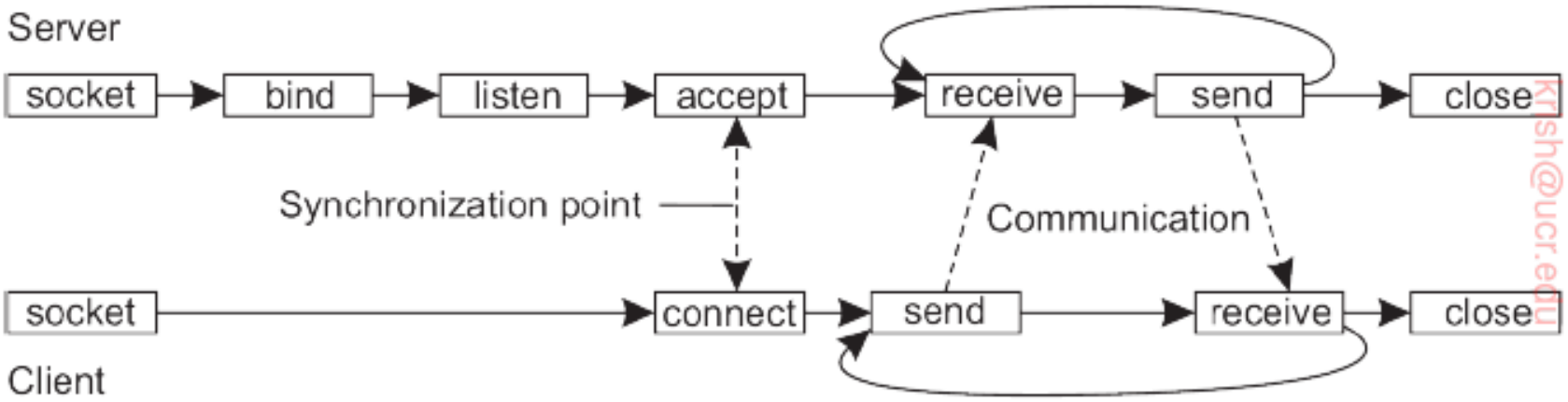
Example: Client

```
1 from socket  import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world')  # send same data
5 data = s.recv(1024)     # receive the response
6 print data               # print the result
7 s.close()               # close the connection
```

Figure 4.20: (b) A simple socket-based client-server system: the client.

- Client reads and prints data that is received

Pictorial description



Limitations

- Sockets simple and elegant
- However, limited and when realizing distributed computations mistakes are possible.
- Also unicast – one to one and relies on TCP (or UDP).

Messaging patterns

- To make network programming easier, we leverage a key observation.
- Messaging applications (or components thereof) typically rely on some simple patterns.
- Enhancing sockets for these patterns can ease development of distributed applications
  - ZeroMQ

Properties of ZeroMQ

- Uses TCP, but application developer does not have to worry about the setting up or maintaining of connections.
- Supports many-to-one communications
  - A socket can be bound to many addresses
  - Server listens to multiple ports using a blocking receive operation.
- Also supports one to many: multicast (later)
- Asynchronous: Sender can continue after submitting message to the underlying communication subsystem

Socket pairing

- ZeroMQ establishes a higher level of abstraction by pairing sockets.
- Each pair corresponds to a specific communication pattern:
  - Request Reply
  - Publish Subscribe
  - Pipeline

Request Reply

- Traditional client-server type communications.
- Client uses a request socket (REQ) to send a request message.
- Server uses a reply socket (of type REP).
- Simplifies matters for developers – they do not need to call listen or accept

Publish Subscribe

- Clients subscribe to specific messages published by servers.
- Only messages to which the client has subscribed are transmitted.
- If server publishes messages to which no one subscribes, they are lost.
- Thus, this is essentially what is called as a multicast from the server to all its subscribing clients.
- Server runs socket type PUB, and client must use SUB type sockets

Example server

```
1  import zmq, time
2
3  context = zmq.Context()
4  s = context.socket(zmq.PUB)          # create a publisher socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.bind(p)                            # bind socket to the address
7  while True:
8    time.sleep(5)                      # wait every 5 seconds
9    s.send("TIME " + time.asctime()) # publish the current time
```

- Naïve time server; publishes its current local time using a PUB socket.
- Local time published every 5 seconds.

Example client

```
1  import zmq
2
3  context = zmq.Context()
4  s = context.socket(zmq.SUB)          # create a subscriber socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.connect(p)                         # connect to the server
7  s.setsockopt(zmq.SUBSCRIBE, "TIME")  # subscribe to TIME messages
8
9  for i in range(5):  # Five iterations
10   time = s.recv()   # receive a message
11   print time
```

- Client creates a SUB socket and connects to the servers PUB socket; if the tag is TIME (this is what the client subscribes to), it retrieves the time and prints five times.

Pipeline

- Process seeks to push out results (as opposed to pulling as in publish/subscribe)
- Does not care about which other process pulls results – the first available is fine.
- A pulling process may get results from multiple processes – will do so from the first pushing process.
- Intent: Keep as many processes working as possible, pushing results through a pipeline of processes as quickly as possible.

Message Passing Interface (MPI)

- Sockets deemed insufficient for two reasons:
  - Wrong level of abstraction (only receive, send; need finer granularity as we will see)
  - Designed for communication over networks : general purpose and not high speed enough.
- Need for a hardware and platform independent interface for communication across a group of processes handling a specific task
  - MPI or Message Passing Interface
  - Takes place within a group
  - Process within a group assigned an identifier (group ID, process ID)

Operations in MPI

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

- Blocking versus non-blocking
  - For example, MPI_ssend has the process blocking until the message transmission starts to the receiver.
  - MPI_recv is blocking but MPI_irecv is non-blocking
  - Can pass pointers (reference) to the outgoing message to prevent need for copies

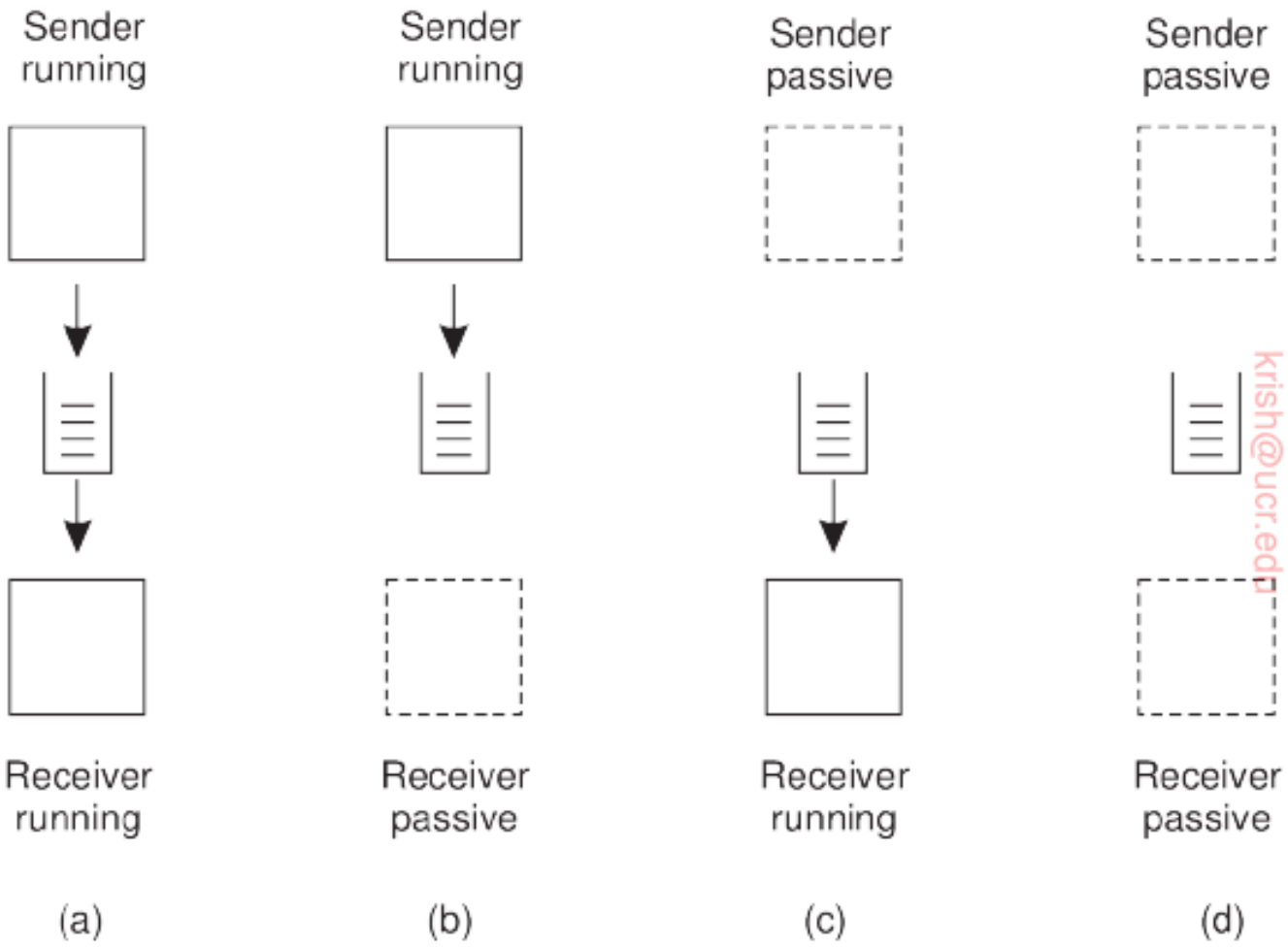
Message Queuing Systems

- Also called Message Oriented Middleware or MOM for short.
- Support for completely asynchronous communications.
- Buffering capacity for messages (storage) even when the sender or receiver is active.
- Persistent and asynchronous : supports message transfers that can take minutes.

Model

- Applications insert messages in specific queues.
- These messages might be forwarded via multiple communication servers to the destination server.
  - If the destination server is down interim, the message is stored at an intermediate server and delivered later
- Each application could have a queue (private) to which other applications send messages.
  - Sharing queues possible but more complex.

Possible combinations



Operations

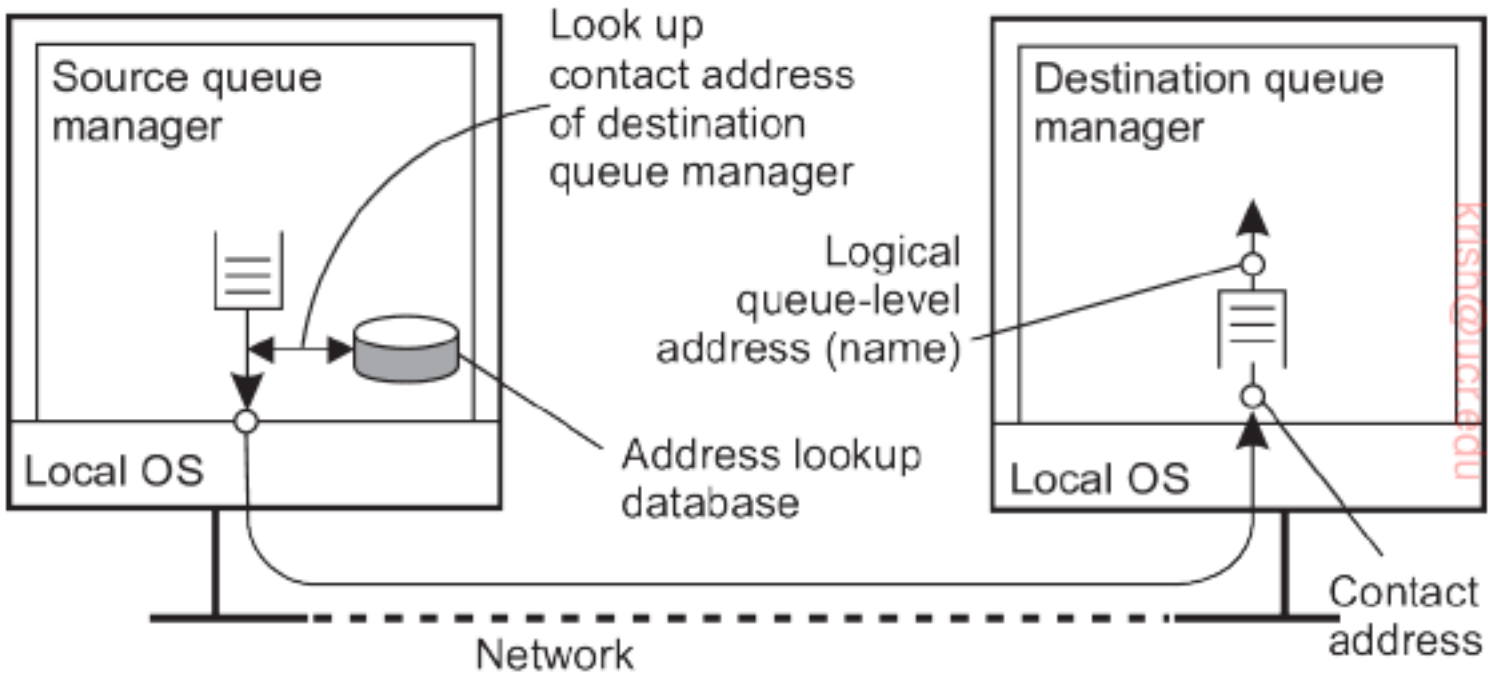
Operation	Description
`put`	Append a message to a specified queue
`get`	Block until the specified queue is nonempty, and remove the first message
`poll`	Check a specified queue for messages, and remove the first. Never block
`notify`	Install a handler to be called when a message is put into the specified queue

- Most queuing systems allow a process to have a handler that is automatically invoked when a message is inserted
- Callbacks can also be used to start the process in such cases.
  - Typically implemented using daemon which continuously monitors receiver queues

Architecture

- Commonly there are what are called queue managers that handle these application centric queues.
- How to name ?
  - Need for a contact address
    - Could use (host, port) but fall back to socket type identifiers.
    - Need naming
  - Map of name to address – lookup table ?
    - When a new queue is added, all tables need to be updated.

Look up depiction



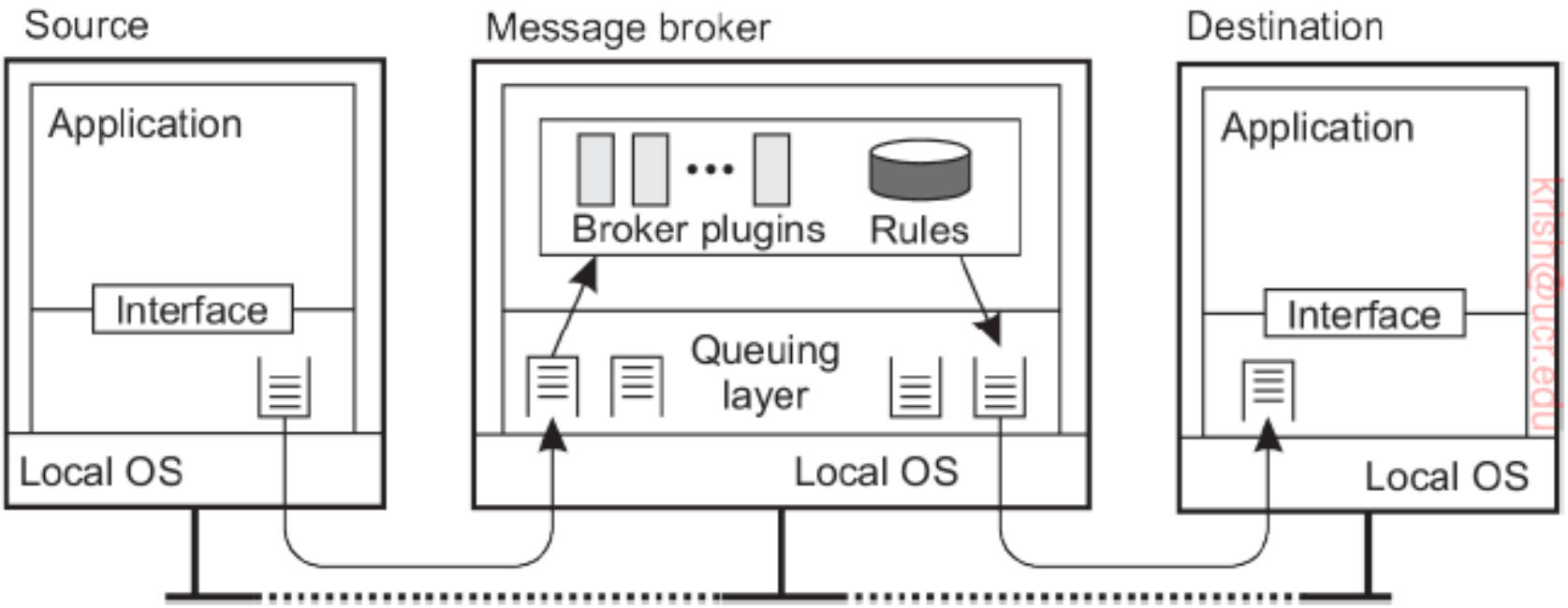
Overlay Network

- The implicit assumption here is that the queue manager of an application A, can directly contact the queue manager of an application B.
  - Not viable at scale
- This would require special queue managers that incorporate routing functionalities.
- So you can envision an application layer network – which is called an “overlay”.

Message Brokers

- Need for translation of message semantics across applications.
- For example, one application may have messages containing a table from a database where:
  - special end-of-record delimiter
  - known fixed length
- Recipient may expect a different delimiter and variable length fields.
- Message brokers perform translations
  - Broker plugins are subprograms that are application specific towards achieving this goal.

Organization of a message broker



IBM WebSphere

- A practical message queuing system
- A wealth of literature (see book and referenced papers).
- All queues are managed by queue managers.
- Each queue manager is responsible for removing messages from its send queues and forwarding to other queue managers.
- Likewise, it picks up messages from the network and stores in appropriate input queue.
- Message sizes have a maximum size (4 MB), and queue (buffer) sizes are specified (2 GB).

Message channels

- Abstractions of transport layer connections
- Unidirectional reliable connection between a sending and receiving queue manager.
  - For example, TCP connection
- The two ends of a message channel are managed by message channel agents (MCAs)
  - Sending MCA: Check send queues, wrap in TCP/IP packet, sending to the receive MCA
  - Receiving MCA: Listen, unwrap and store unwrapped message in appropriate queue.

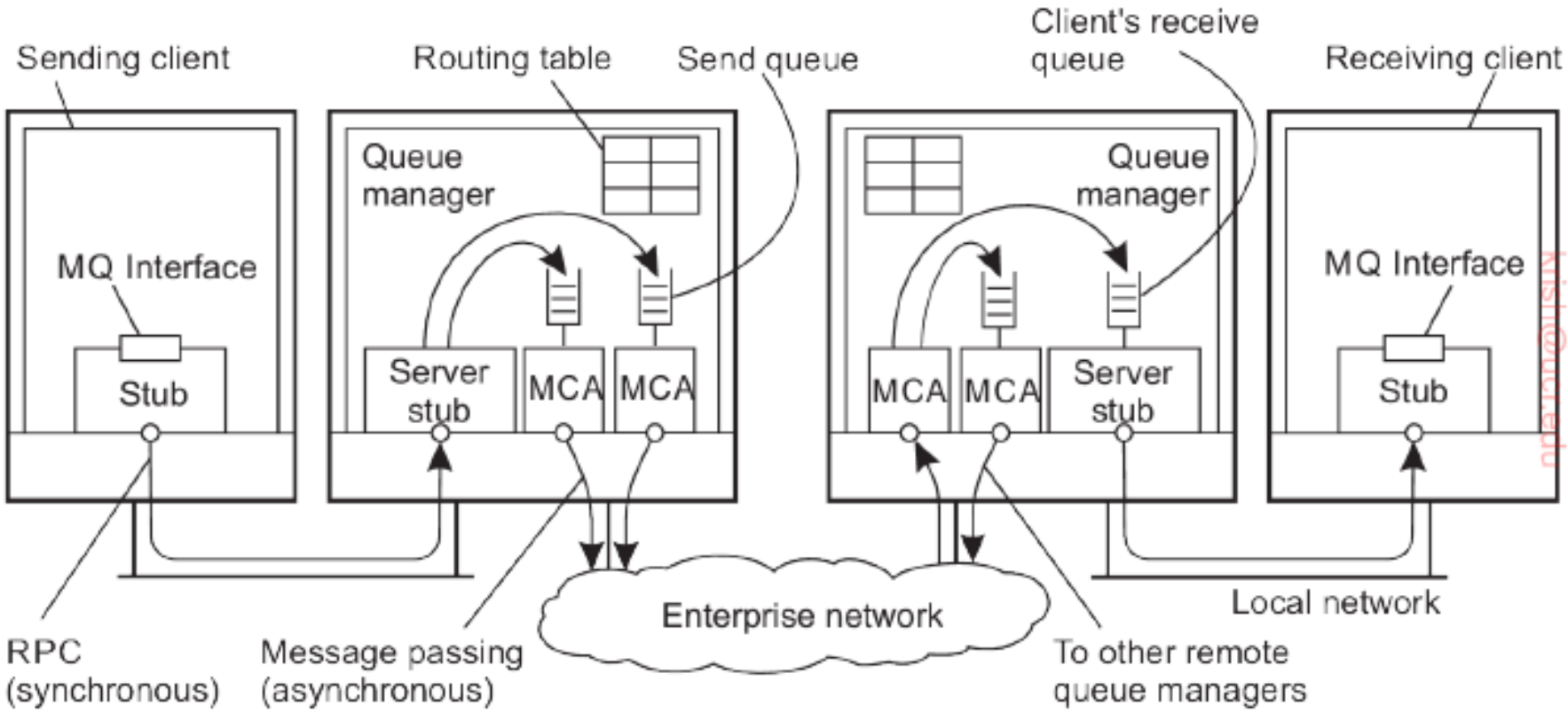
Message channels (cont)

- Both sending and receiving MCAs must be up and running.
- When the message is first put in the queue, a trigger is set off – which invokes a handler to start the sending MCA.
- One can also start an MCA over the network – a control message to a daemon that is listening on the other side.
- Channels are stopped automatically when no messages dropped in queues for a pre-specified time.

MCA Attributes

- Attribute values of sending and receiving MCA should be compatible.
- One way is to negotiate these before the channel set up
  - Same transport protocol
  - FIFO ordering of messages (inserted in send queue, picked up from receive queue)
  - Maximum message length

General organization depiction



- Application uses RPC to communicate with the queue manager based on synchronous communication if the latter is on a different machine.

Addressing and routing

- Address in MQ consists of two parts
  - Name of queue manager to which the message is to be delivered.
  - Destination queue under that manager to which the message is to be appended.
- In addition, routes are to be specified.
- Use of routing tables – the entry specifies the local send queue to which the message is to be appended.
  - This specifies which queue manager the message is to be forwarded.

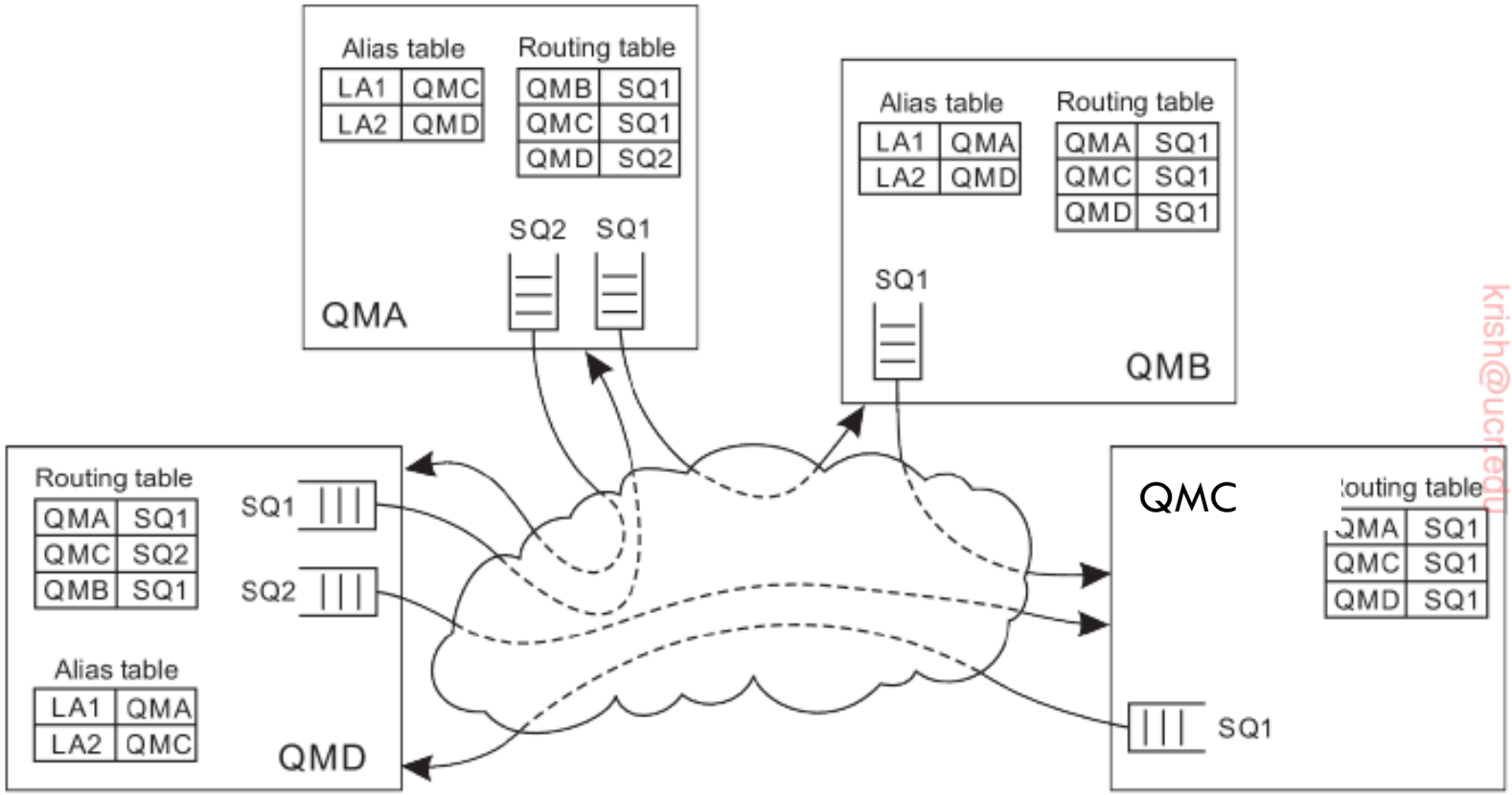
Routing tables

- Routes are explicitly stored in a queue manager using routing tables.
- Each entry is a pair (destQM, sendQ)
  - destQM → destination queue manager
  - sendQ → name of local send queue
- A routing table entry is called an alias in MQ.
- When an intermediate (not destination) QM gets a message, it extracts the name of the destQM and does a routing table look up to find which send queue is to be used (to append the message).

Alias table

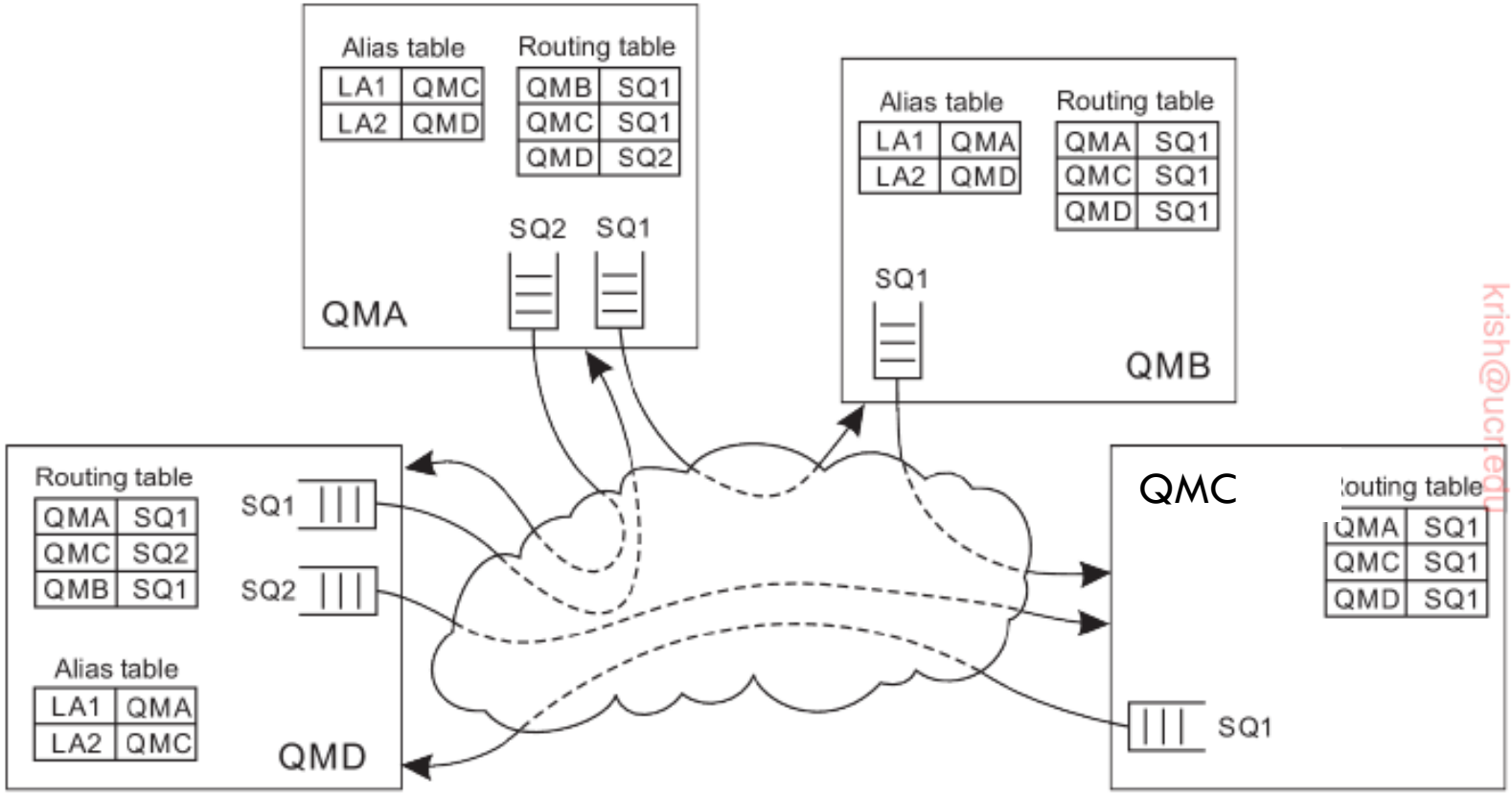
- Each queue manager has a systemwide unique name.
- But changing this name, will affect all applications that send messages to it.
- To alleviate problems a local alias for queue manager names is isued.
- An alias defined with a QM (say QM1) is another name for a different queue manager (QM2), but is available only to applications interfacing with QM1.
  - Alias allows the use of the same (logical) name for a queue even if that queue name changes.

Example



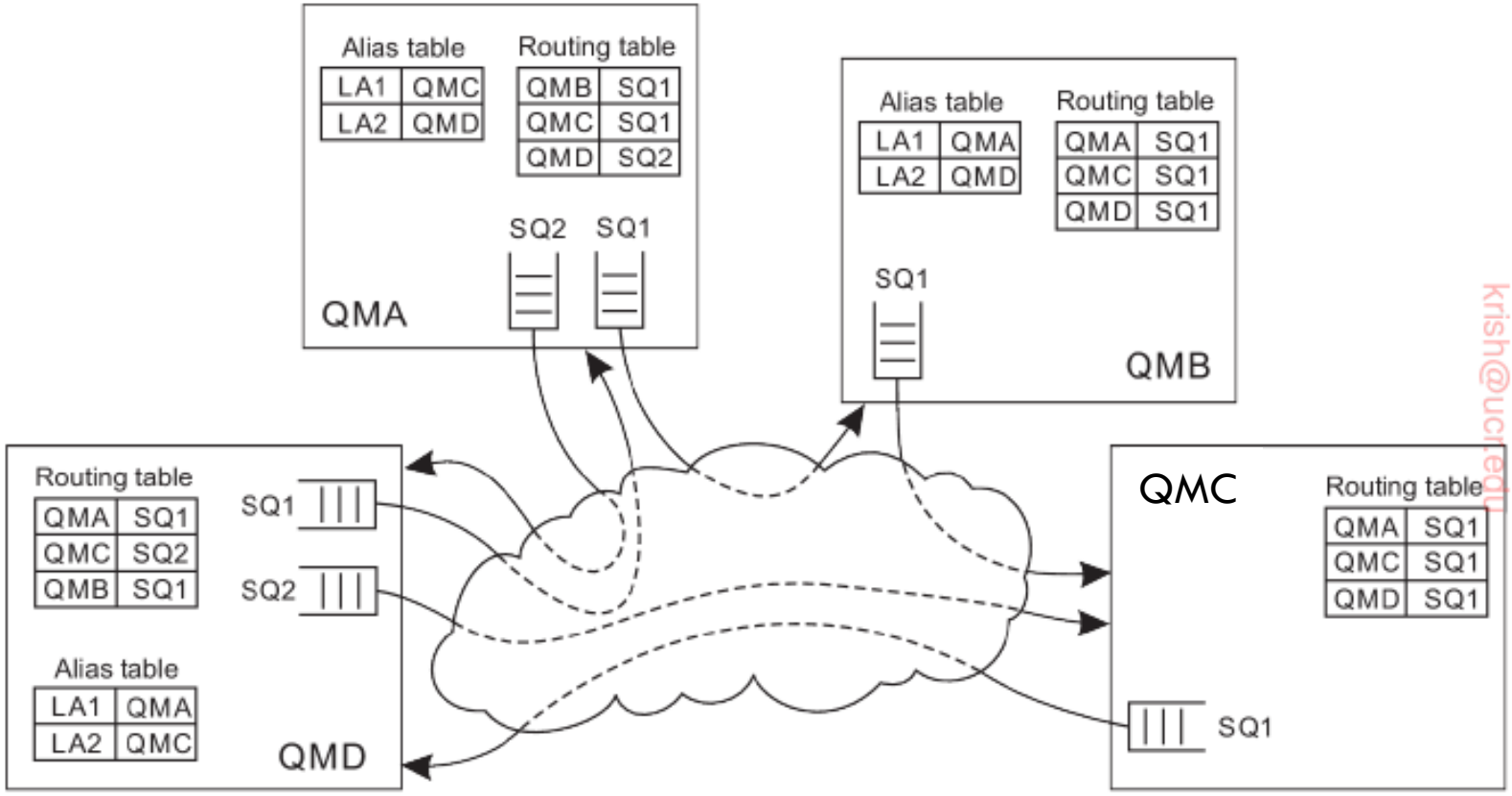
Let us assume an application linked to QMA wants to send a message. It can refer to a remote QM (QMC) refering to an alias LA1.

Example (2)



QMA looks up the alias and maps it to QMC. Route to QMC is found in the routing table and indicates message to be appended to SQ1.

Example (3)



This causes messages to be transferred to QMB. QMB uses its routing table to insert message in SQ1 to be sent to QMC.