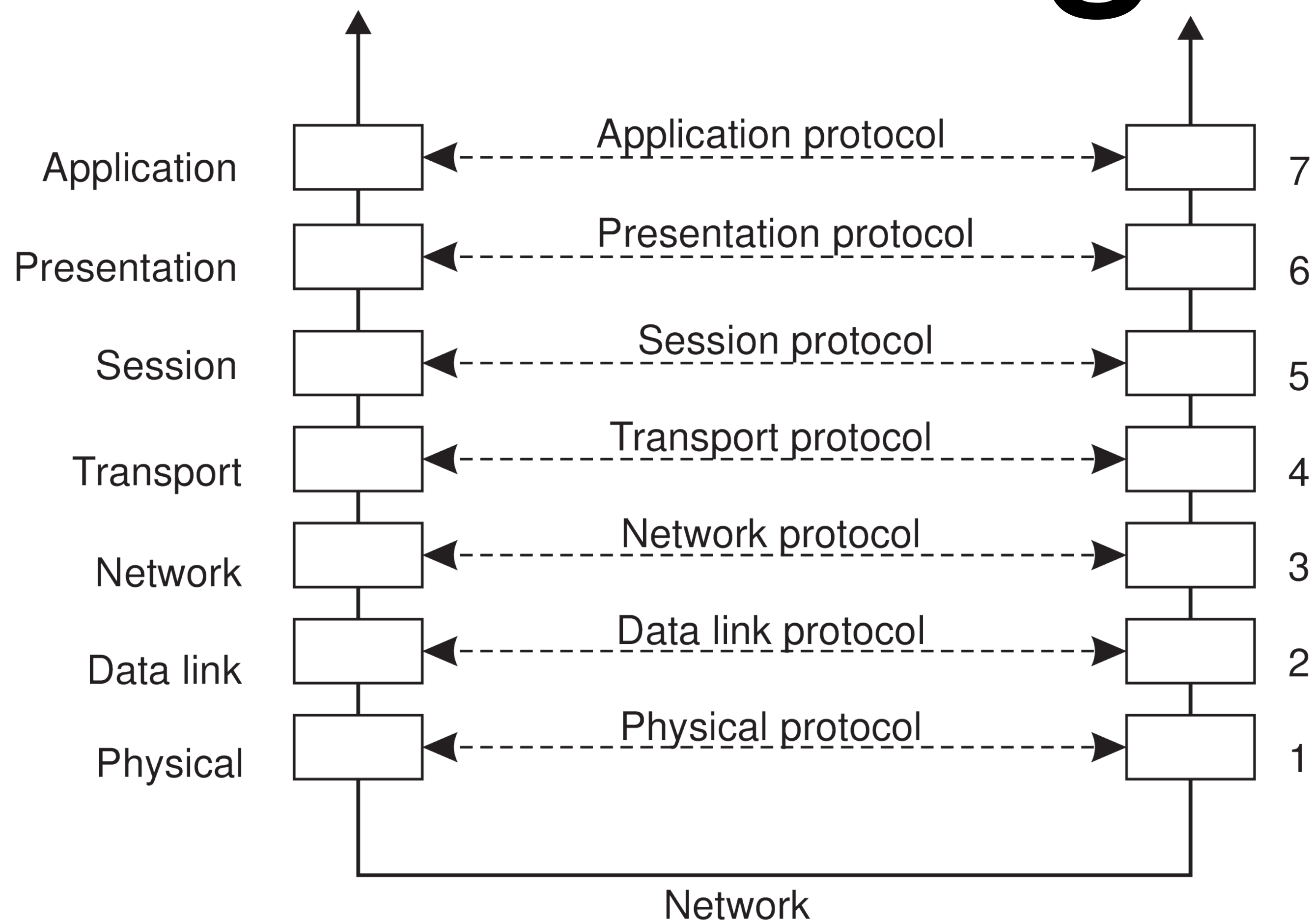
Communication

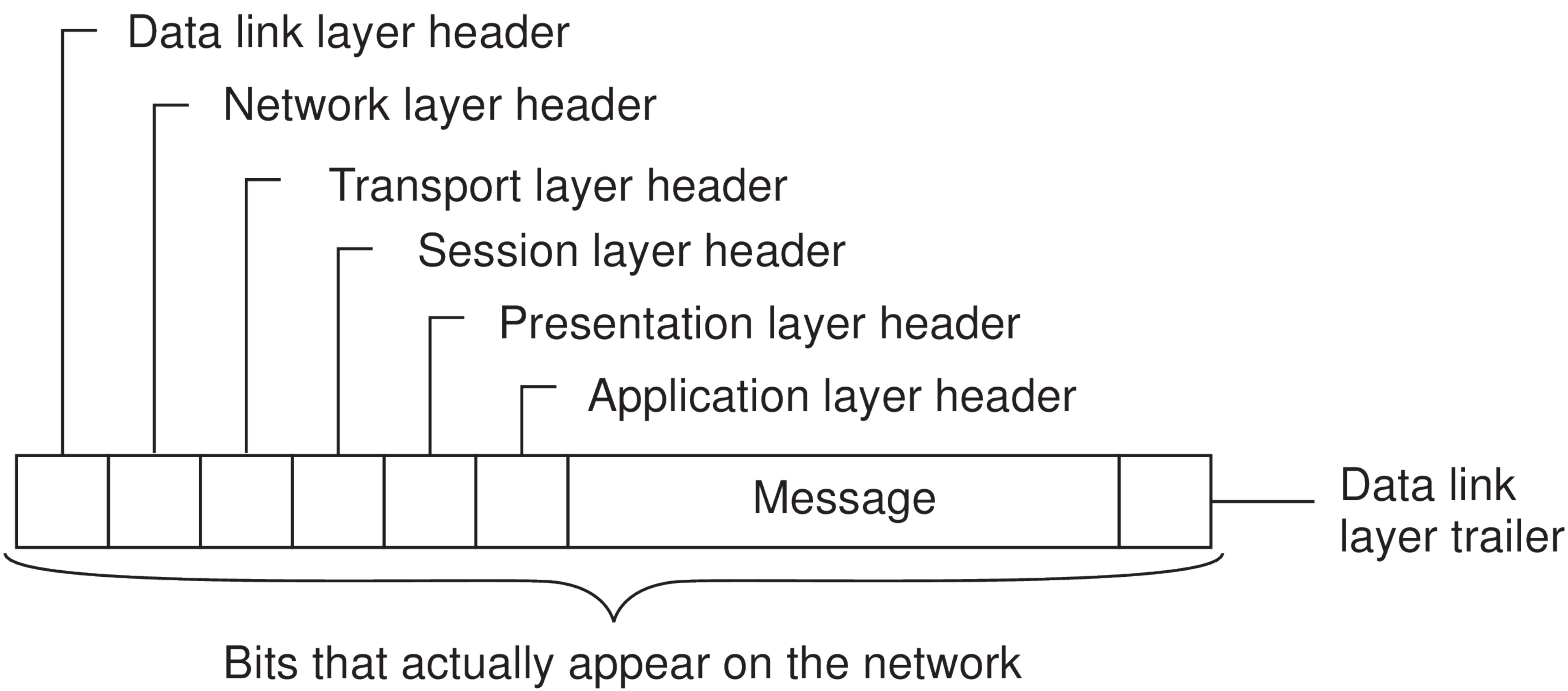
Thomas Schwarz, SJ

Basic Networking Model



- Drawbacks
  - Focus on message-passing only
  - Often unneeded or unwanted functionality
  - Violates access transparency

Encapsulation of Messages



Low-level layers

- Recap
  - Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver
  - Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control
  - Network layer: describes how packets in a network of computers are to be routed.

- Observation
  - For many distributed systems, the lowest-level interface is that of the network layer.

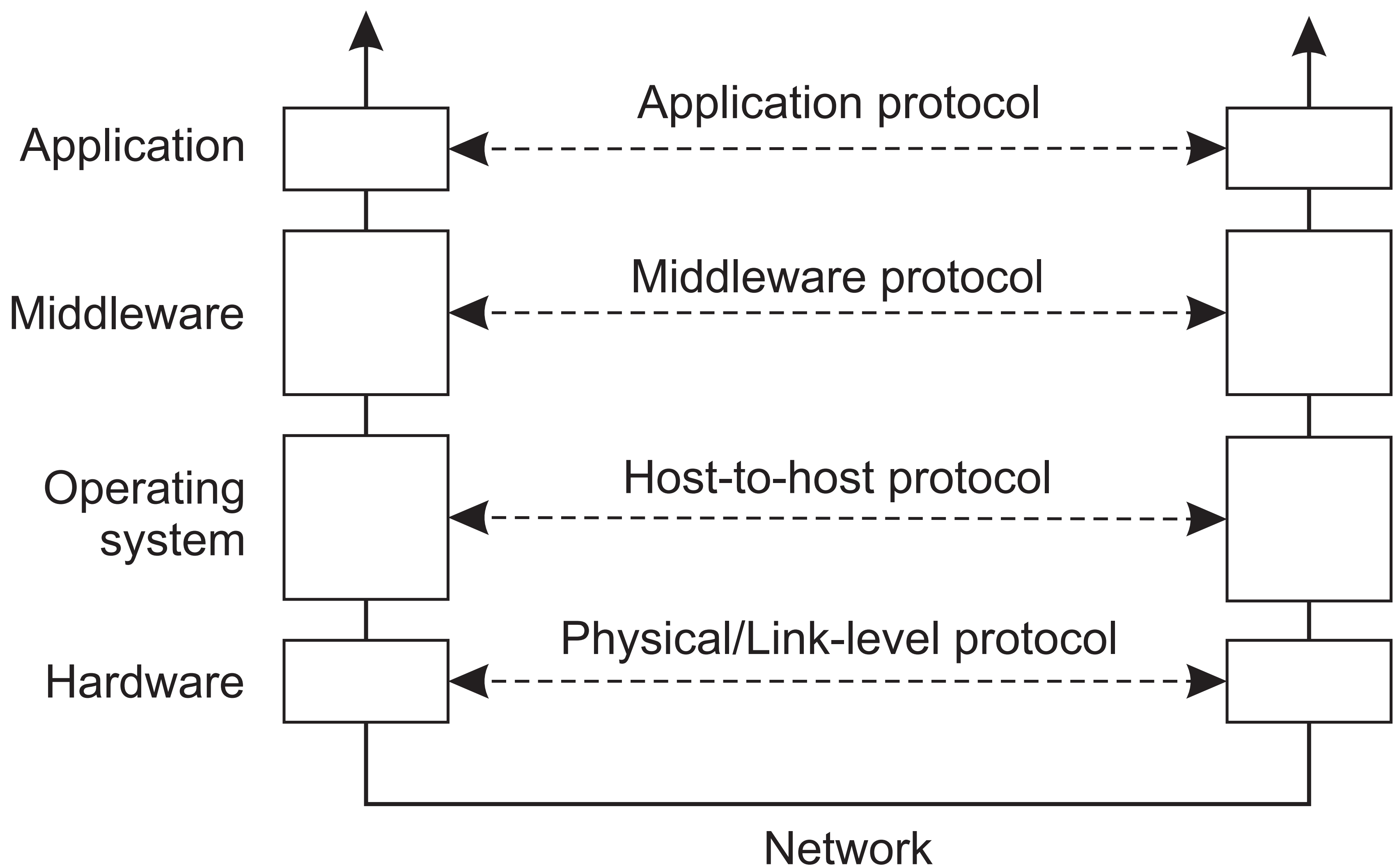
Transport Layer

- Important
  - The transport layer provides the actual communication facilities for most distributed systems.
  - Standard Internet protocols
    - TCP: connection-oriented, reliable, stream-oriented communication
    - UDP: unreliable (best-effort) datagram communication

Middleware layer

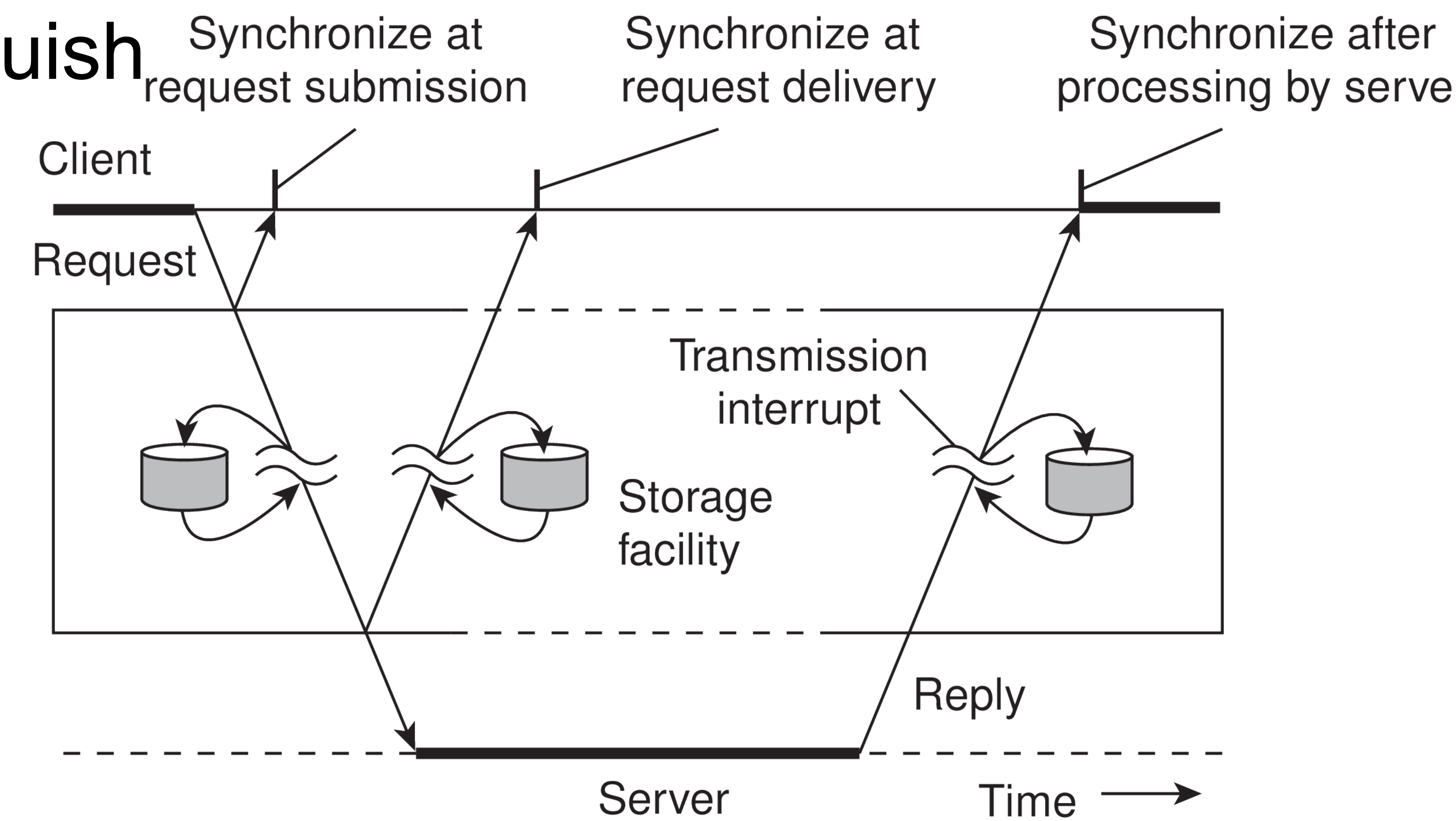
- Middleware is invented to provide common services and protocols that can be used by many different applications
  - A rich set of communication protocols
  - (Un)marshaling of data, necessary for integrated systems
  - Naming protocols, to allow easy sharing of resources
  - Security protocols for secure communication
  - Scaling mechanisms, such as for replication and caching

An adapted layering model



Types of communication

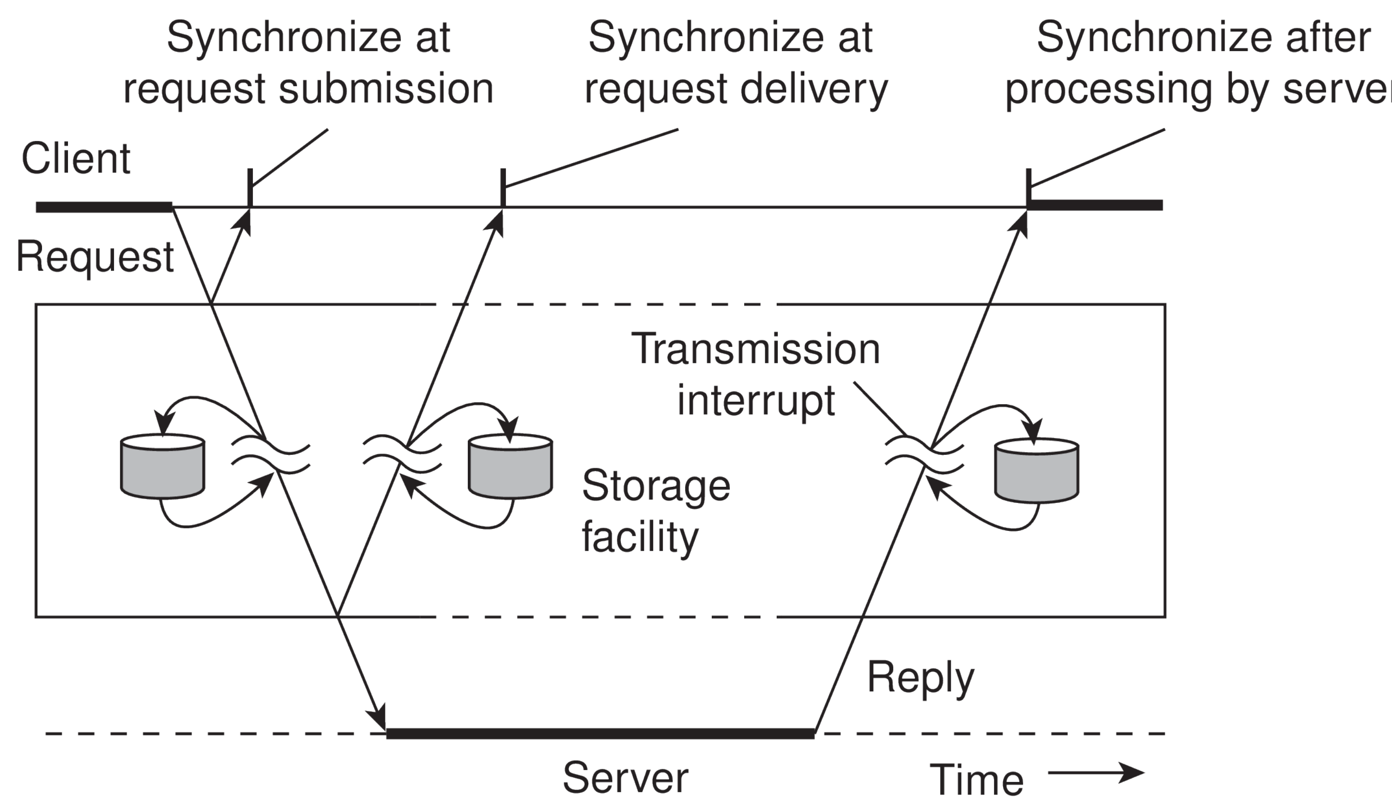
Distinguish



- Transient versus persistent communication
- Asynchronous versus synchronous communication

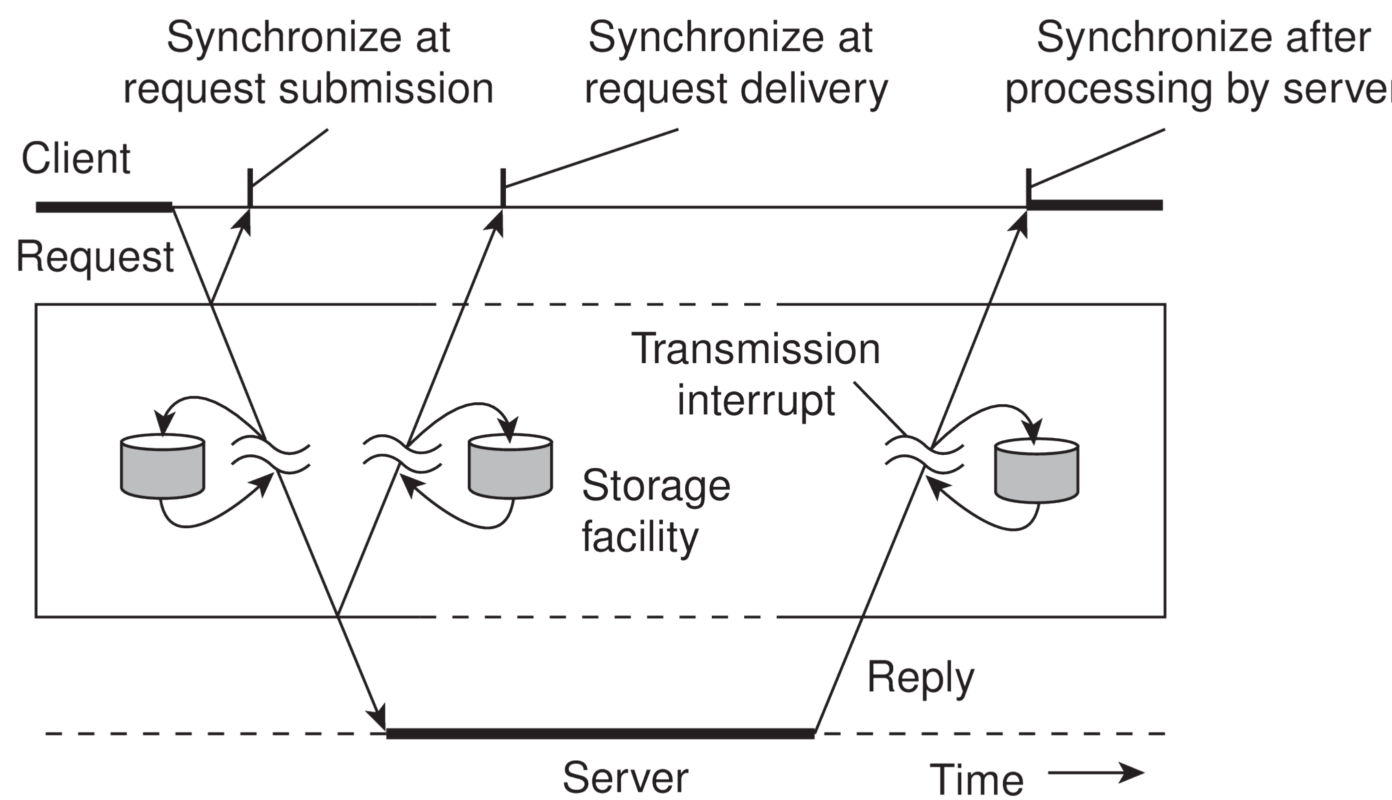
Types of communication

Distinguish



- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.

Types of communication



Synchronize

- At request submission
- At request delivery
- After request processing

Client-Server

- Some observations
  - Client/Server computing is generally based on a model of transient synchronous communication:
    - Client and server have to be active at the time of communication
    - Client issues request and blocks until it receives reply
    - Server essentially waits only for incoming requests, and subsequently processes them

Client-Server

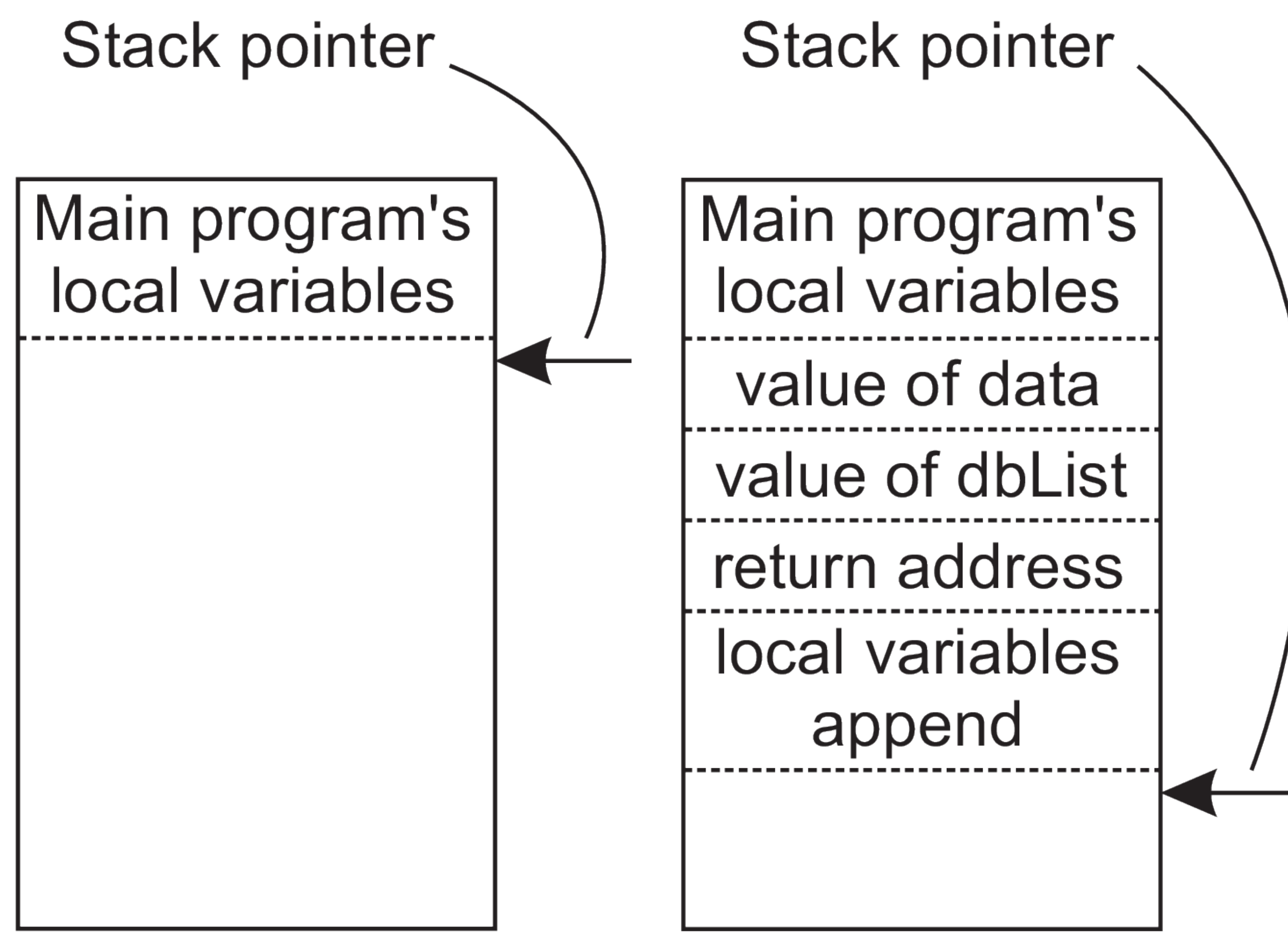
- Drawbacks of synchronous communication
  - Client cannot do any other work while waiting for reply
  - Failures have to be handled immediately: the client is waiting
  - The model may simply not be appropriate (mail, news)

Messaging

- Message-oriented middleware
- Aims at high-level persistent asynchronous communication:
  - Processes send each other messages, which are queued
  - Sender need not wait for immediate reply, but can do other things
  - Middleware often ensures fault tolerance

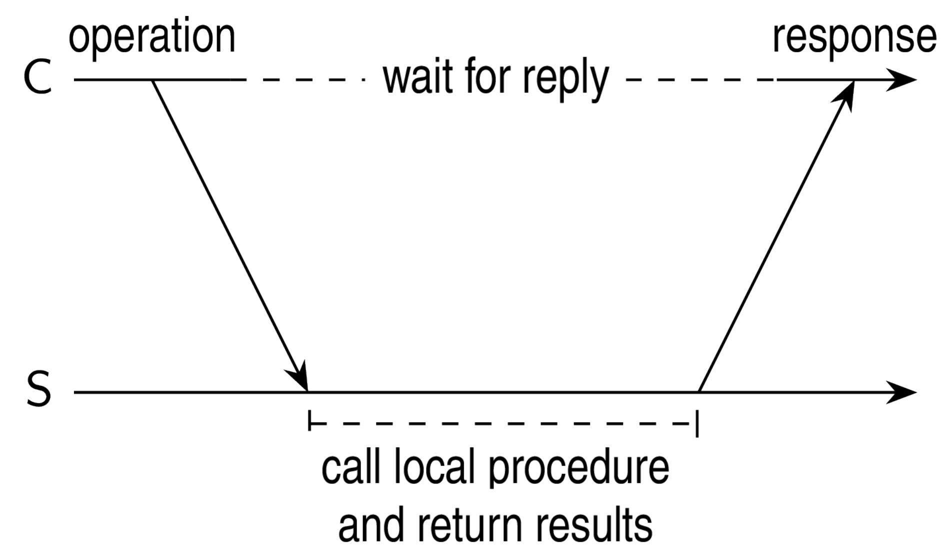
Remote Procedure Calls

- Local procedure calls
  - call by value
  - call by reference
  - call by copy
    - (rare)
- Passing using
  - registers
  - stacks

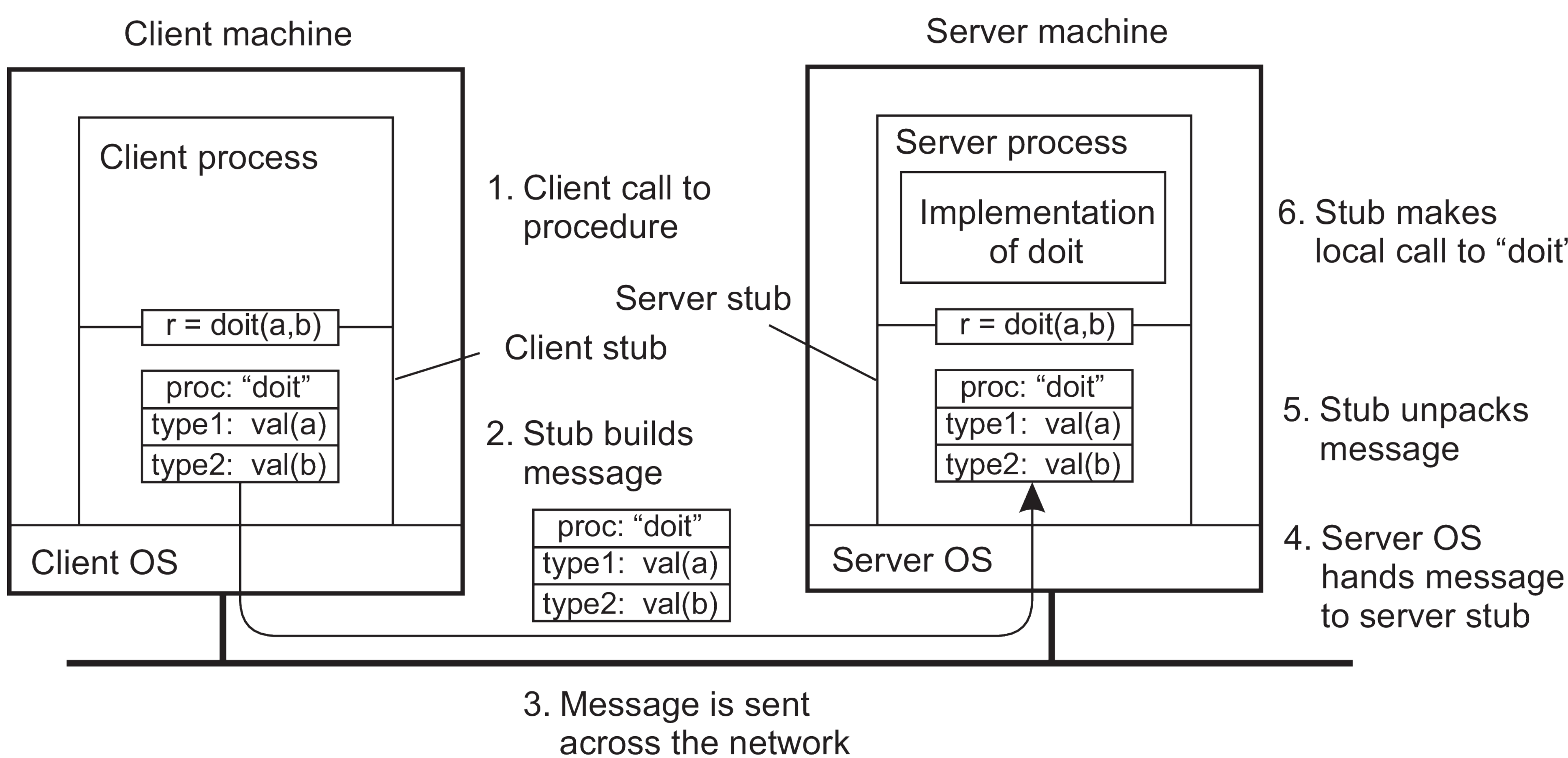


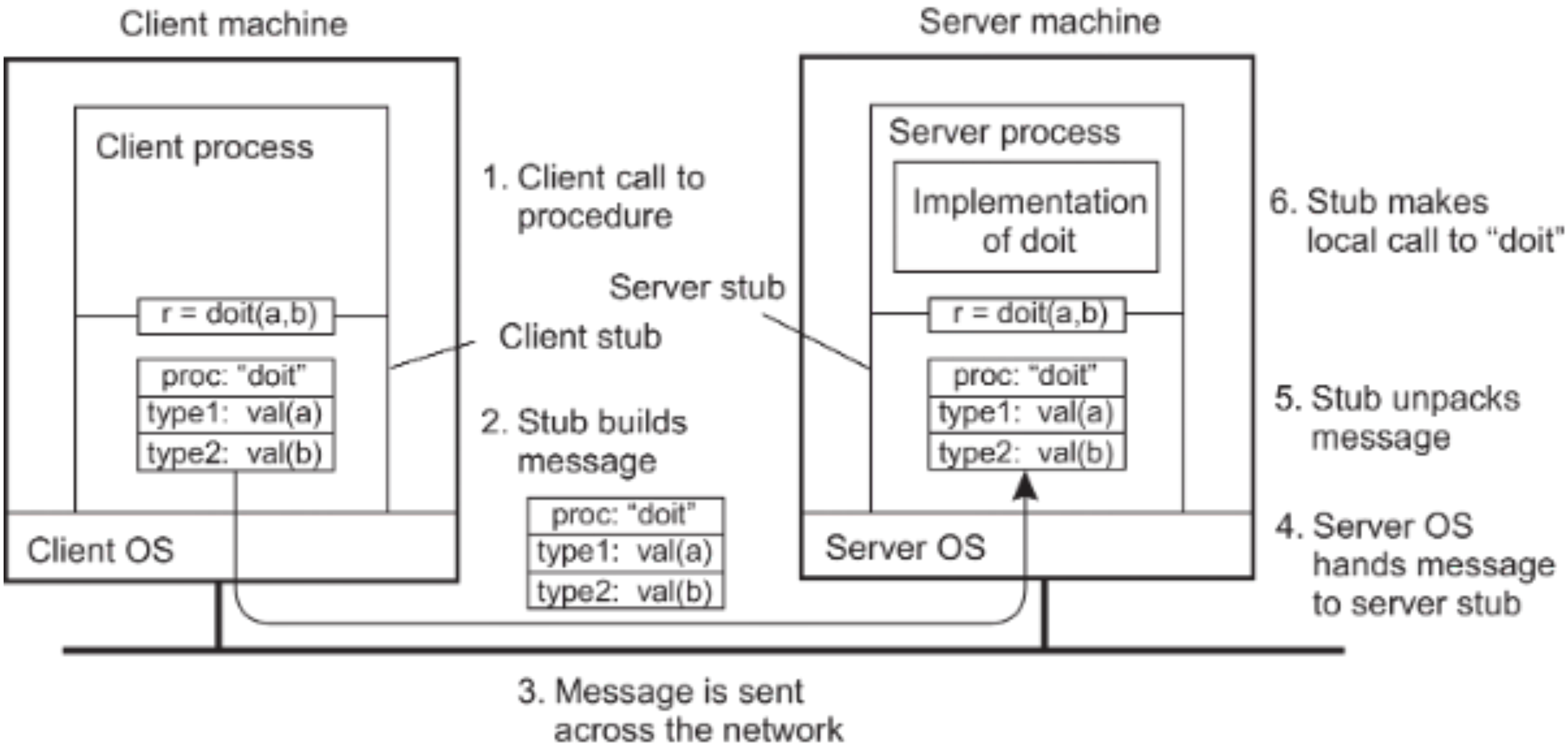
Remote Procedure Calls

- Observations
  - Application developers are familiar with simple procedure model
  - Well-engineered procedures operate in isolation (black box)
  - There is no fundamental reason not to execute procedures on separate machine
- Conclusion
  - Communication between caller & callee can be hidden by using procedure-call mechanism.



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 does local call; returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result; returns to client.

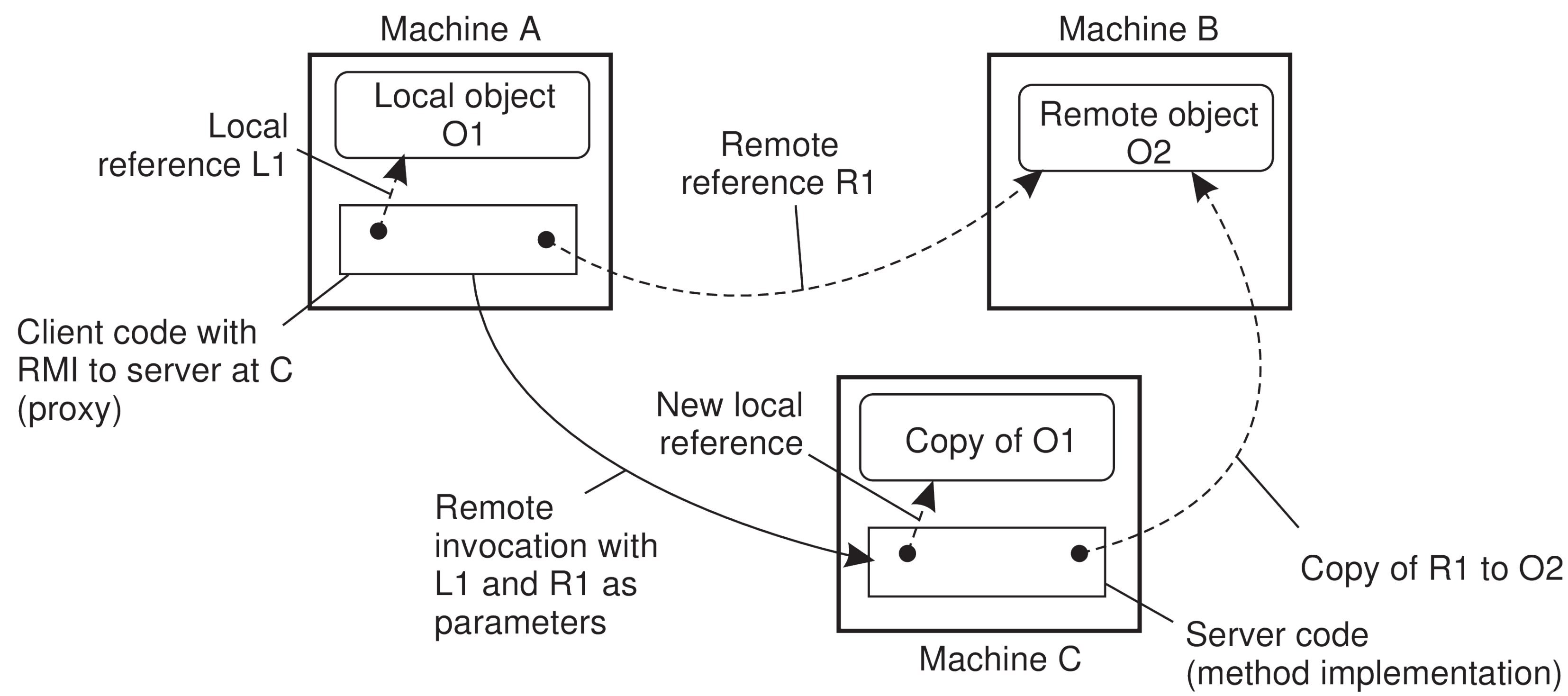
Basic RPC Operations

- RPC Parameter passing
  - There's more than just wrapping parameters into a message
  - Client and server machines may have different data representations (think of byte ordering)
  - Wrapping a parameter means transforming a value into a sequence of bytes
  - 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)

- Conclusion
- Client and server need to properly interpret messages, transforming them into machine-dependent representations

Basic RPC Operations

- Parameter passing in object based systems
  - Object references to local and remote objects are treated differently



Client program runs on Machine A
Server runs on Machine C

RPC makes reference to object O1 in machine A and O2 in machine B

Copy O1 (pass by value) and reference to O2 (pass by reference)

Basic RPC Operations

- Some assumptions
  - Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
  - All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.
- Conclusion
  - Full access transparency cannot be realized

RPC Parameter Passing

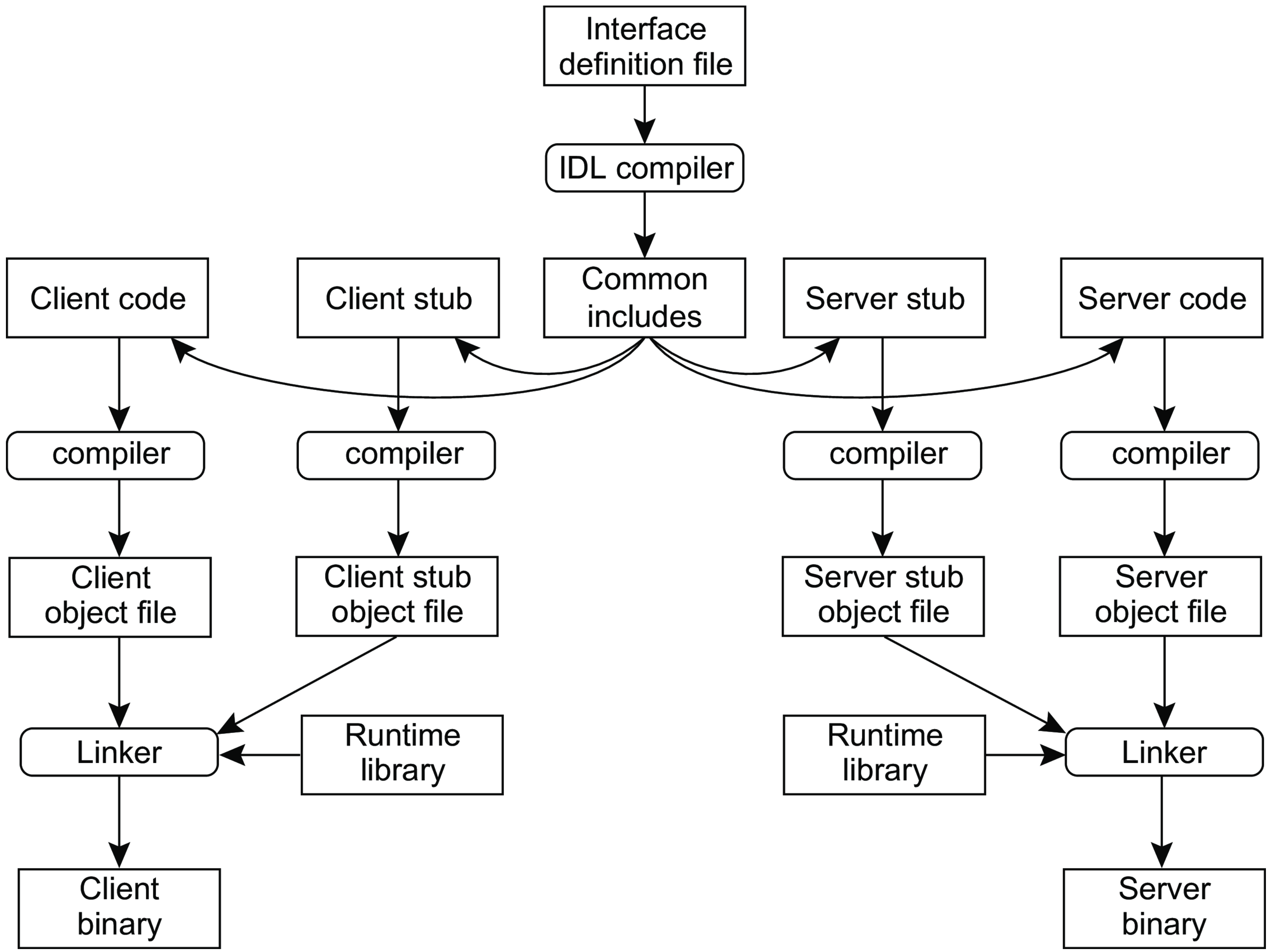
- There's more than just wrapping parameters into a message
  - Client and server machines may have different data representations (think of byte ordering)
  - Wrapping a parameter means transforming a value into a sequence of bytes
  - Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)

- Conclusion
  - Client and server need to properly interpret messages, transforming them into machine-dependent representations.

RPC Application Support

- Both sides in a RPC call need to follow the same conventions
  - Need to implement stubs
  - Often use an ***I*nterface *D*efinition *L*anguage (IDL)**

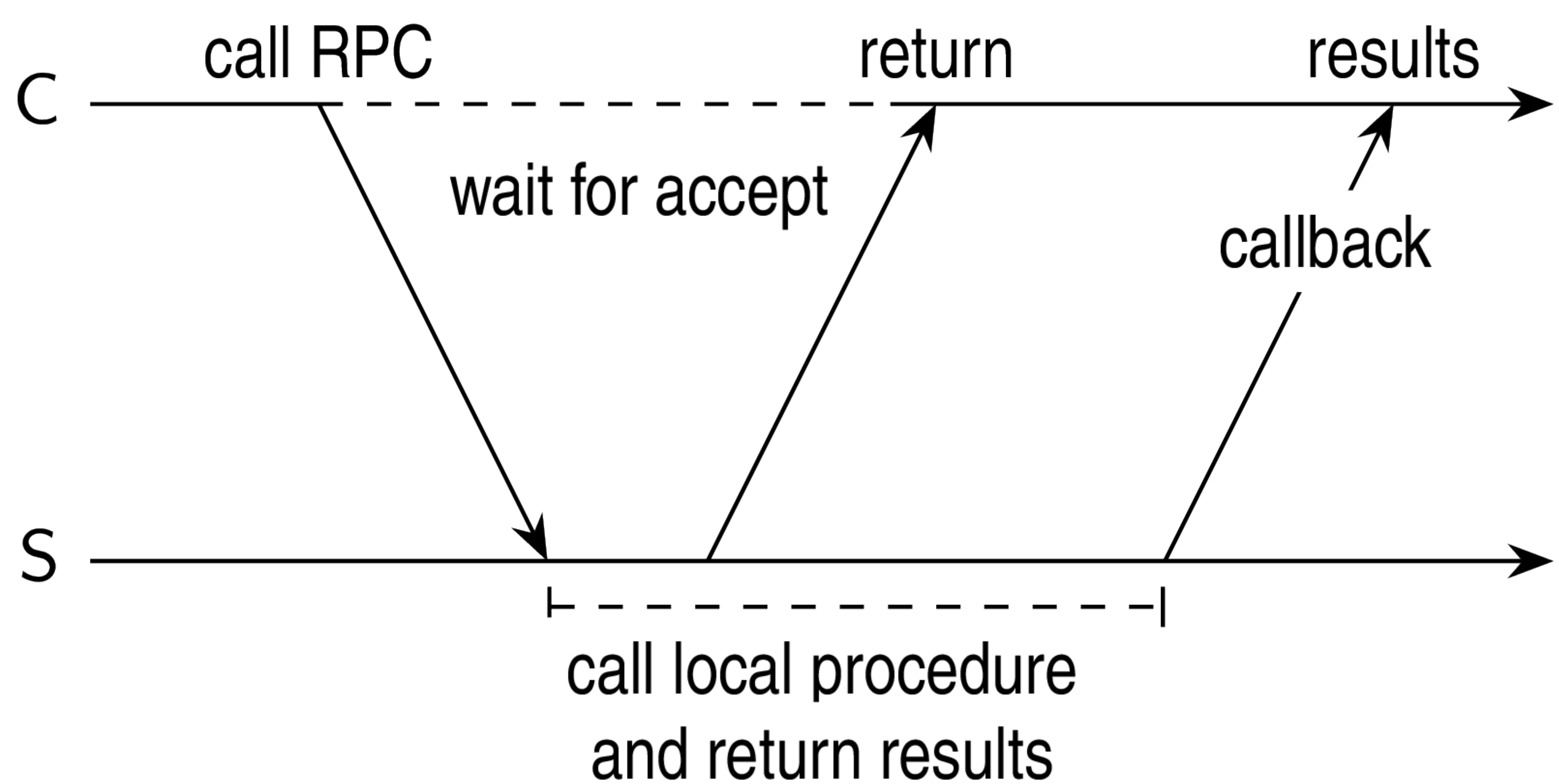
RPC Application Support



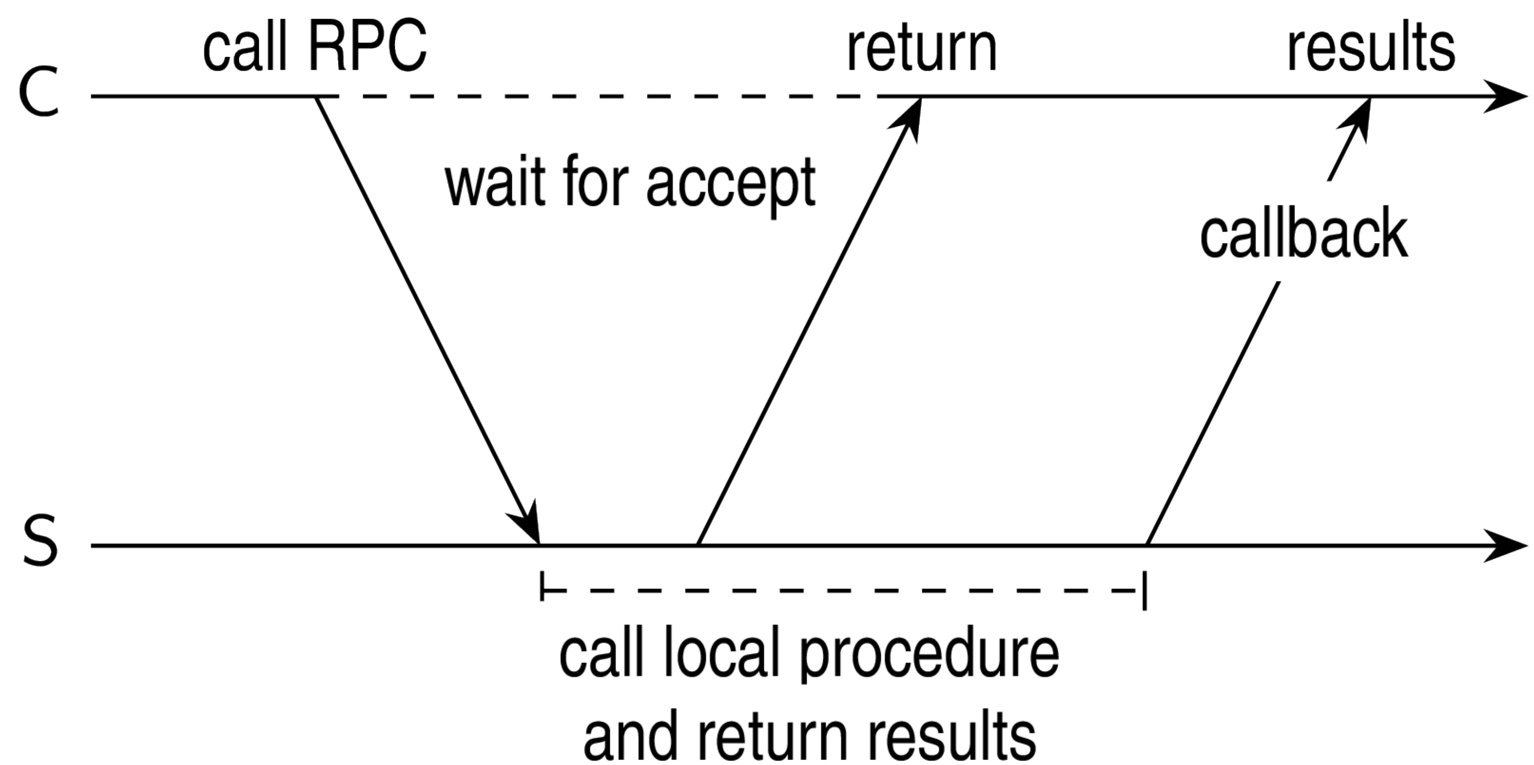
Generating stubs from an IDL file

Asynchronous RPC

- Essence
  - Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.



Asynchronous RPC

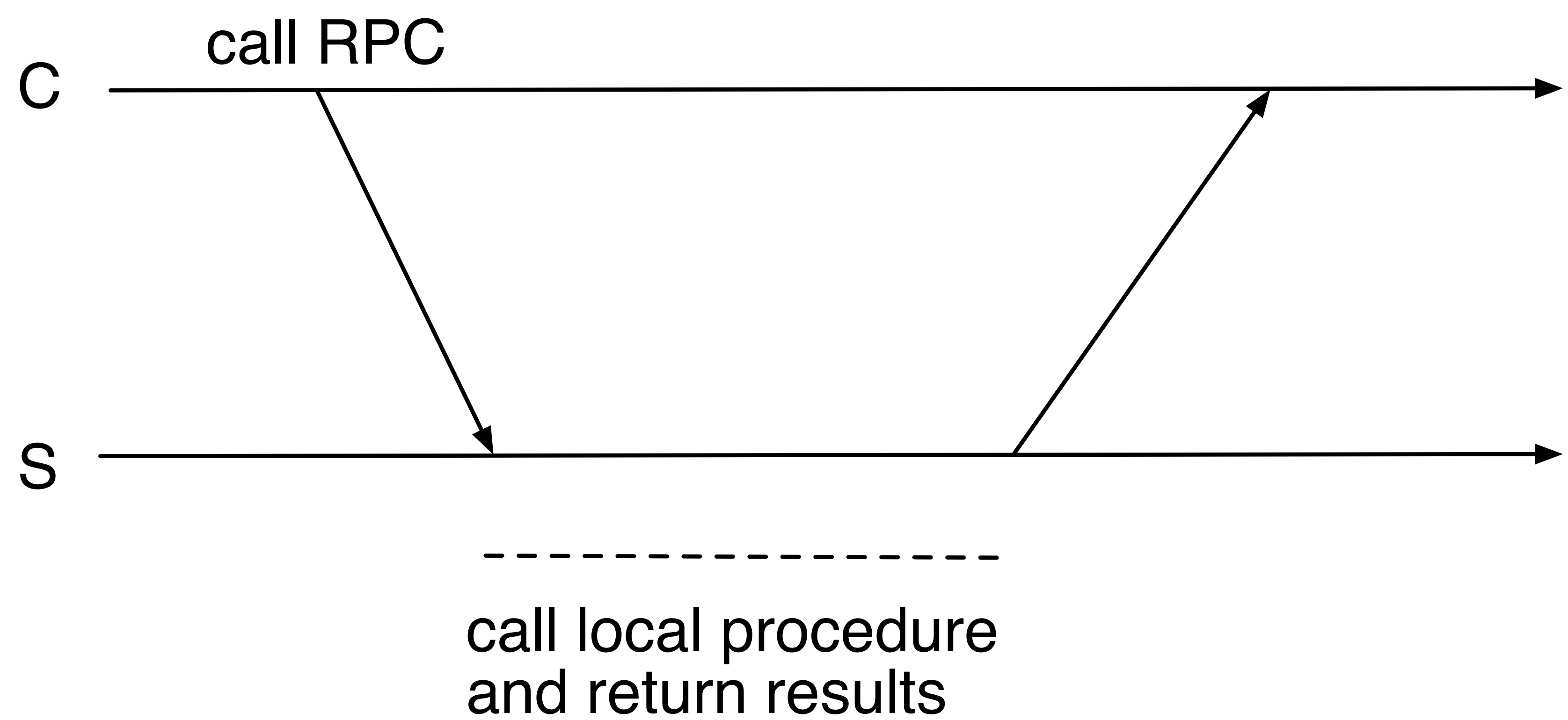


Deferred synchronous RPC:

Clients calls Server and waits for acceptance and continues
Server upon completion sends a message
Client makes a ***callback***

Callbacks are user-defined functions invoked when an event happens

Asynchronous RPC

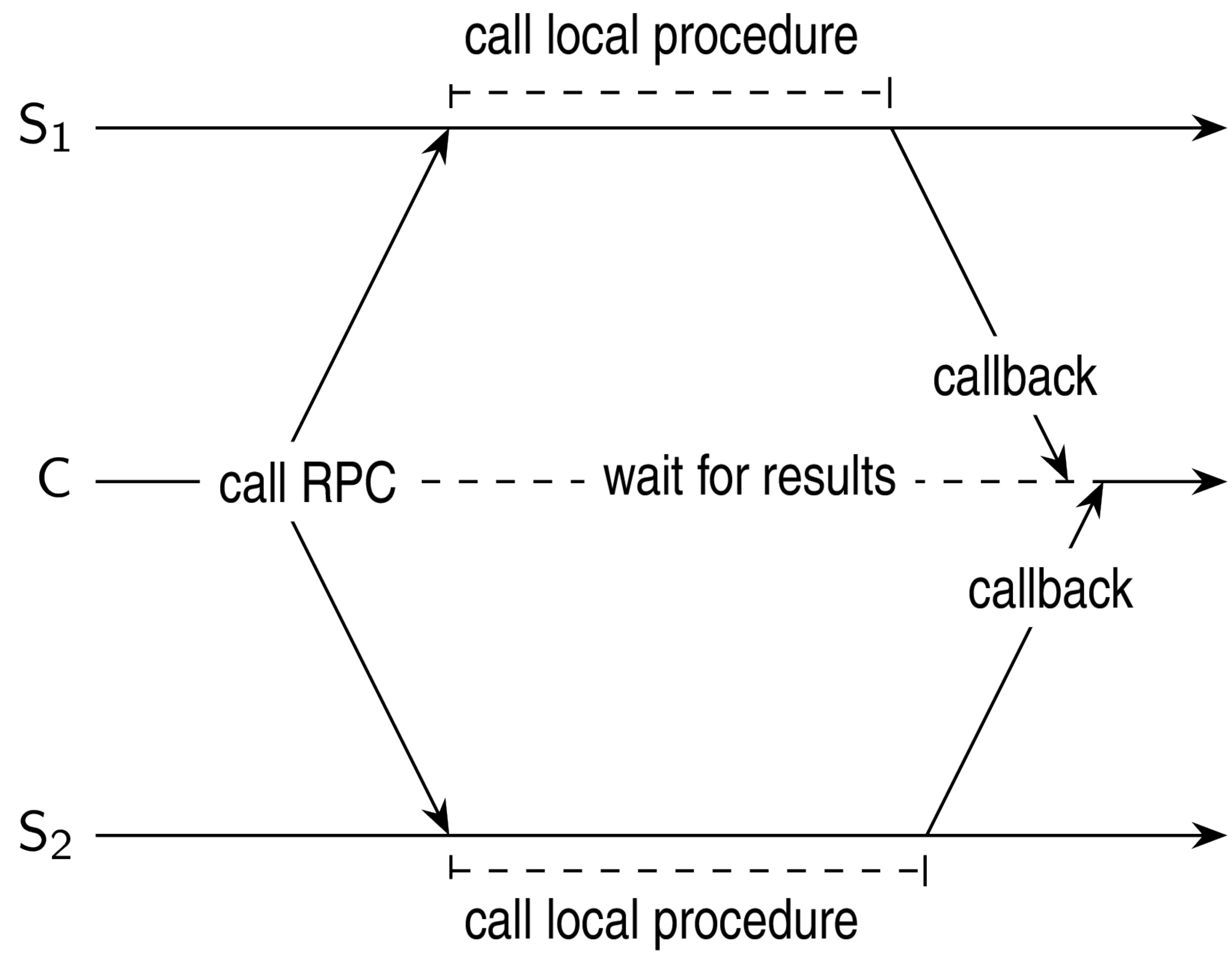


One way RCP

Client continues after call
Server sends a message or client polls server

Multicast RPC

- Does client know that there is more than one RPC call?
- When does the caller react?
    - Wait for the first or for the last value?

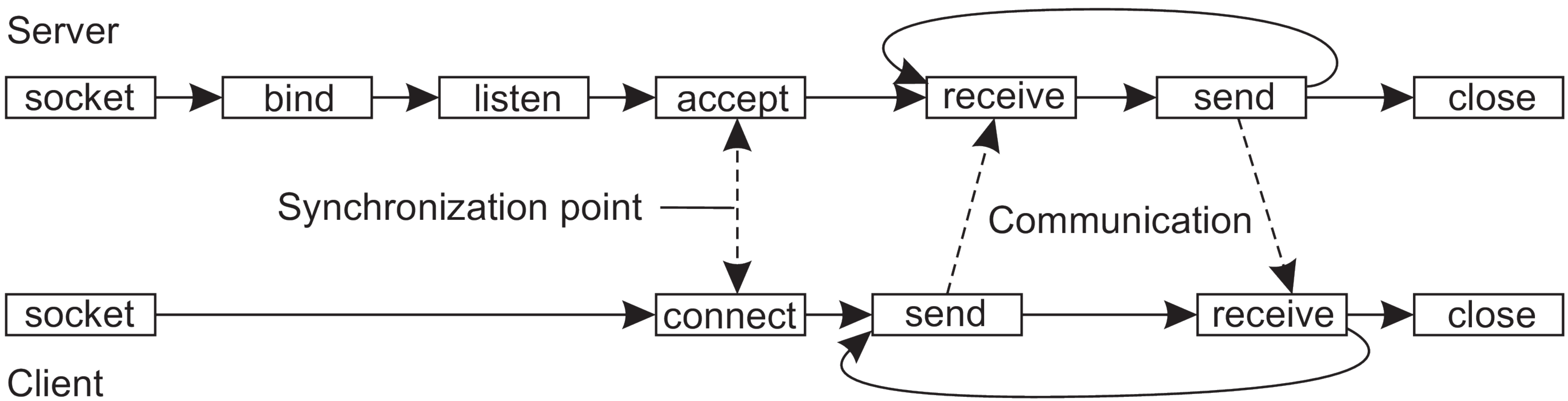


Message Oriented Communication

- Simple transient messages with sockets

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

Message Oriented Communication



Message Oriented Communication

- Advanced transient messaging
  - Overcome the brittleness of sockets
  - ZeroMQ (2011)
    - Provides a higher level of expression by pairing sockets: one for sending messages at process P and a corresponding one at process Q for receiving messages. All communication is asynchronous.
      - Three patterns
    - Request-reply; Publish-subscribe; Pipeline

Message Oriented Communication

Client uses a Request Socket

Server uses a Response Socket

(no listen no accept)

```
import zmq

def server():
  context = zmq.Context()
  socket  = context.socket(zmq.REP)       # create reply socket
  socket.bind("tcp://*:12345")            # bind socket to address

  while True:
    message = socket.recv()               # wait for incoming message
    if not "STOP" in str(message):        # if not to stop...
      reply = str(message.decode())+'*'   # append "*" to message
      socket.send(reply.encode())         # send it away (encoded)
    else:
      break                               # break out of loop and end

def client():
  context = zmq.Context()
  socket  = context.socket(zmq.REQ)       # create request socket

  socket.connect("tcp://localhost:12345") # block until connected
  socket.send(b"Hello world")             # send message
  message = socket.recv()                 # block until response
  socket.send(b"STOP")                    # tell server to stop
  print(message.decode())                 # print result
```

Message Oriented Communication

Publish-Subscribe patterns

Server publishes a “time server” on a publishing socket

Clients creates a subscribe socket

```
import multiprocessing
import zmq, time

def server():
  context = zmq.Context()
  socket = context.socket(zmq.PUB)            # create a publisher socket
  socket.bind("tcp://*:12345")                # bind socket to the address
  while True:
    time.sleep(5)                             # wait every 5 seconds
    t = "TIME " + time.asctime()
    socket.send(t.encode())                   # publish the current time

def client():
  context = zmq.Context()
  socket = context.socket(zmq.SUB)            # create a subscriber socket
  socket.connect("tcp://localhost:12345")     # connect to the server
  socket.setsockopt(zmq.SUBSCRIBE, b"TIME")   # subscribe to TIME messages

  for i in range(5):      # Five iterations
    time = socket.recv()  # receive a message related to subscription
    print(time.decode())  # print the result
```

Message Oriented Communication

Producer-worker pattern or pipeline pattern:

Process wants to push out results and others want to pull them

First available worker will pick up work from the producer

If there are free workers, one will be provided with a task.

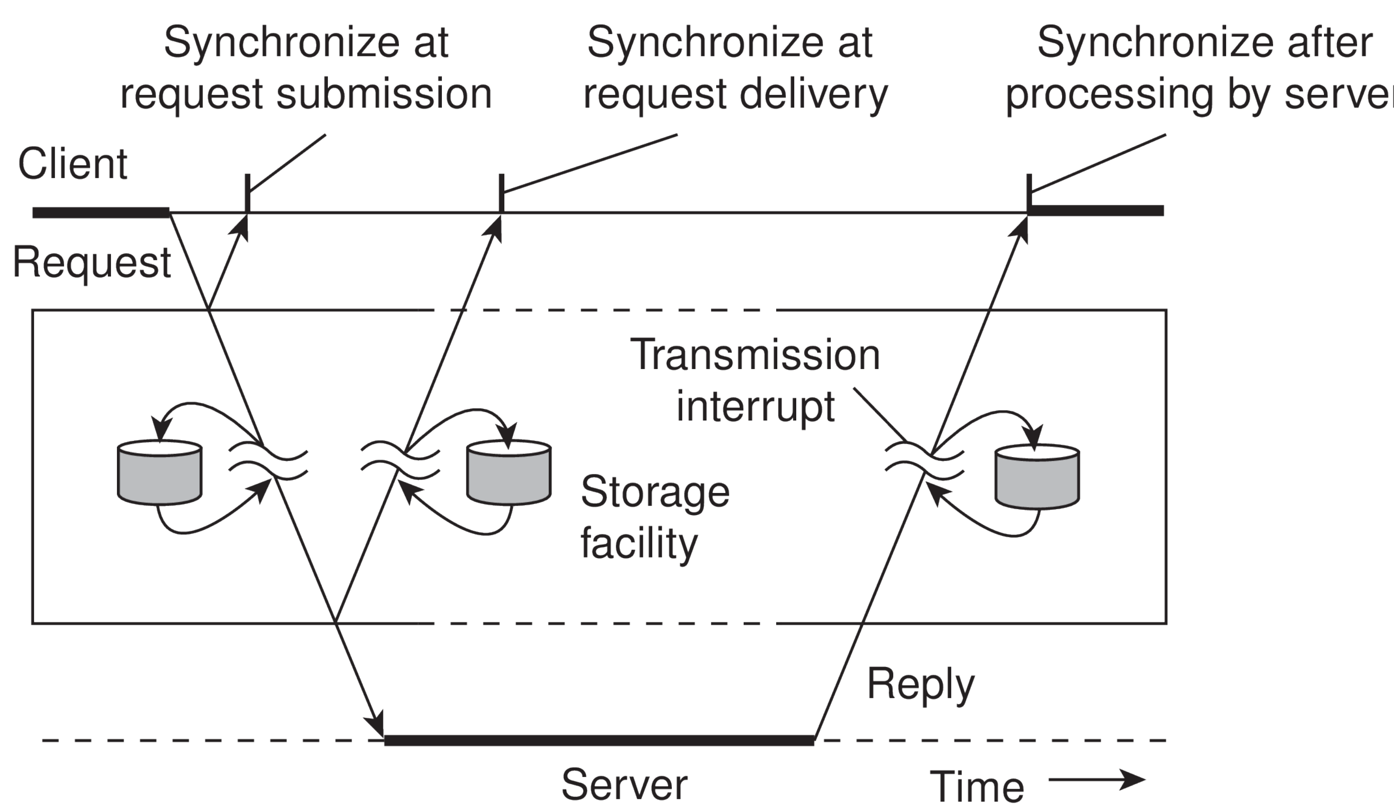
```
def producer():
  context = zmq.Context()
  socket  = context.socket(zmq.PUSH)      # create a push socket
  socket.bind("tcp://127.0.0.1:12345")    # bind socket to address

  while True:
    workload = random.randint(1, 100)     # compute workload
    socket.send(pickle.dumps(workload))   # send workload to worker
    time.sleep(workload/NWORKERS)         # balance production by waiting

def worker(id):
  context = zmq.Context()
  socket  = context.socket(zmq.PULL)      # create a pull socket
  socket.connect("tcp://localhost:12345") # connect to the producer

  while True:
    work = pickle.loads(socket.recv())     # receive work from a source
    time.sleep(work)                       # pretend to work
```

Message Oriented Communication

- ***M**essage **P**assing **I**nterface (MPI)*
    - Sockets only support simple send and receive messages
    - Communicate using general purpose protocol stacks (TCP/IP)
    - Solution should be platform independent

Message Oriented Communication

- MPI:
  - Forms middleware layer
    - With buffers, …



Message Oriented Communication

- MPI:
  - Designed for parallel processing
    - Uses transient communications
  - Serious failures are fatal (no recovery)
  - Processes has an identifier: (groupID, processID)

Message Oriented Communication

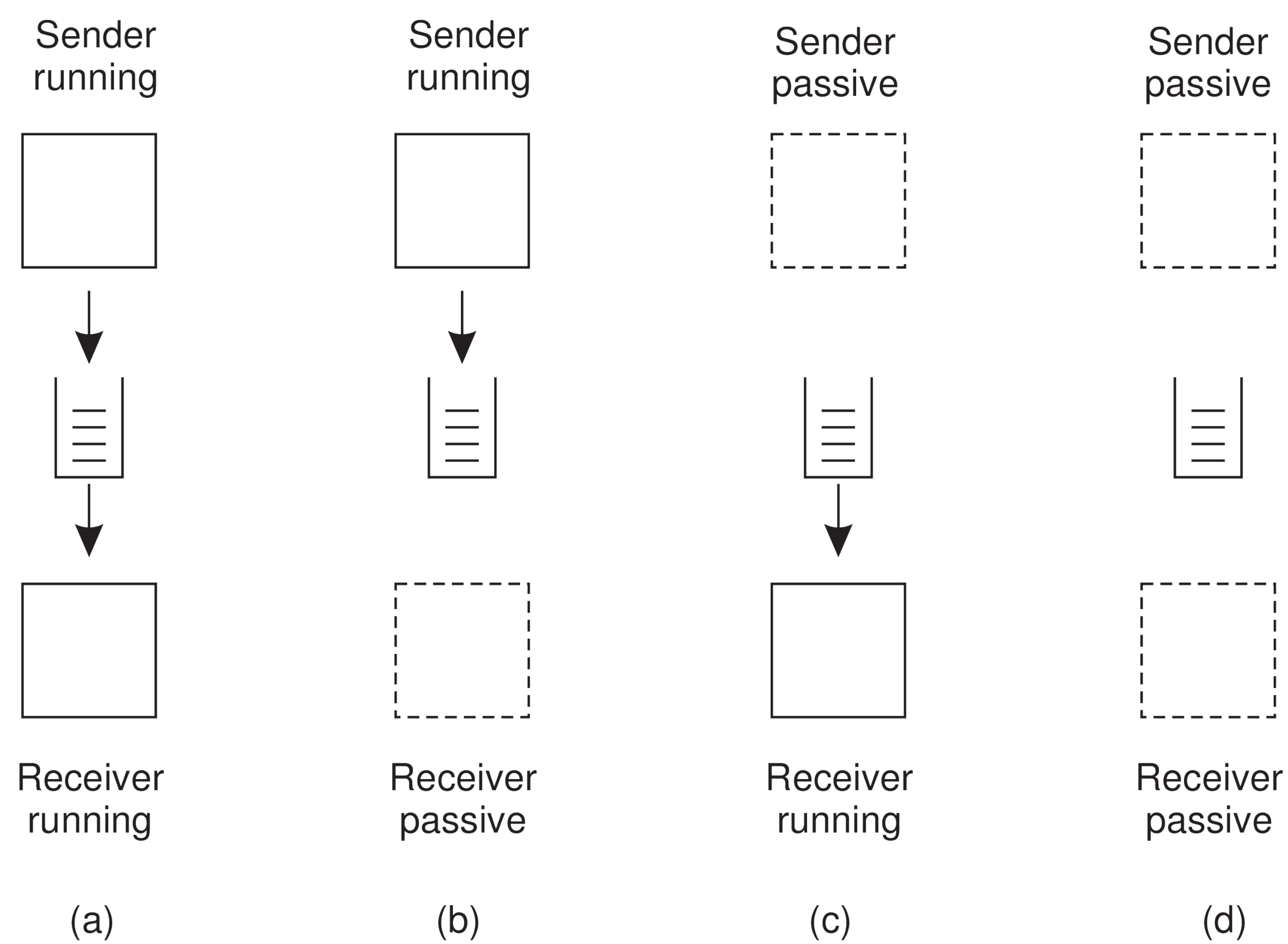
Operation	Description
MPI_BSEND	Append outgoing message to a local send buffer
MPI_SEND	Send a message and wait until copied to local or remote buffer
MPI_SSEND	Send a message and wait until transmission starts
MPI_SENDRECV	Send a message and wait for reply
MPI_ISEND	Pass reference to outgoing message, 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

Message Oriented Communication

- Message-oriented persistent communication
    - Message Queuing Systems
    - Message Oriented Middleware (MOM)
- provides support for persistent asynchronous communication
-

Message Oriented Communication

- Applications communicate by inserting messages in specific queues
  - There is no guarantee that a message will be read by the recipient



Message Oriented Communication

- Queue interface

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

Message Oriented Communication

- Can install handler as a callback function
  - Automatically invoked whenever a message is put into the queue
  - Use a NOTIFY operation

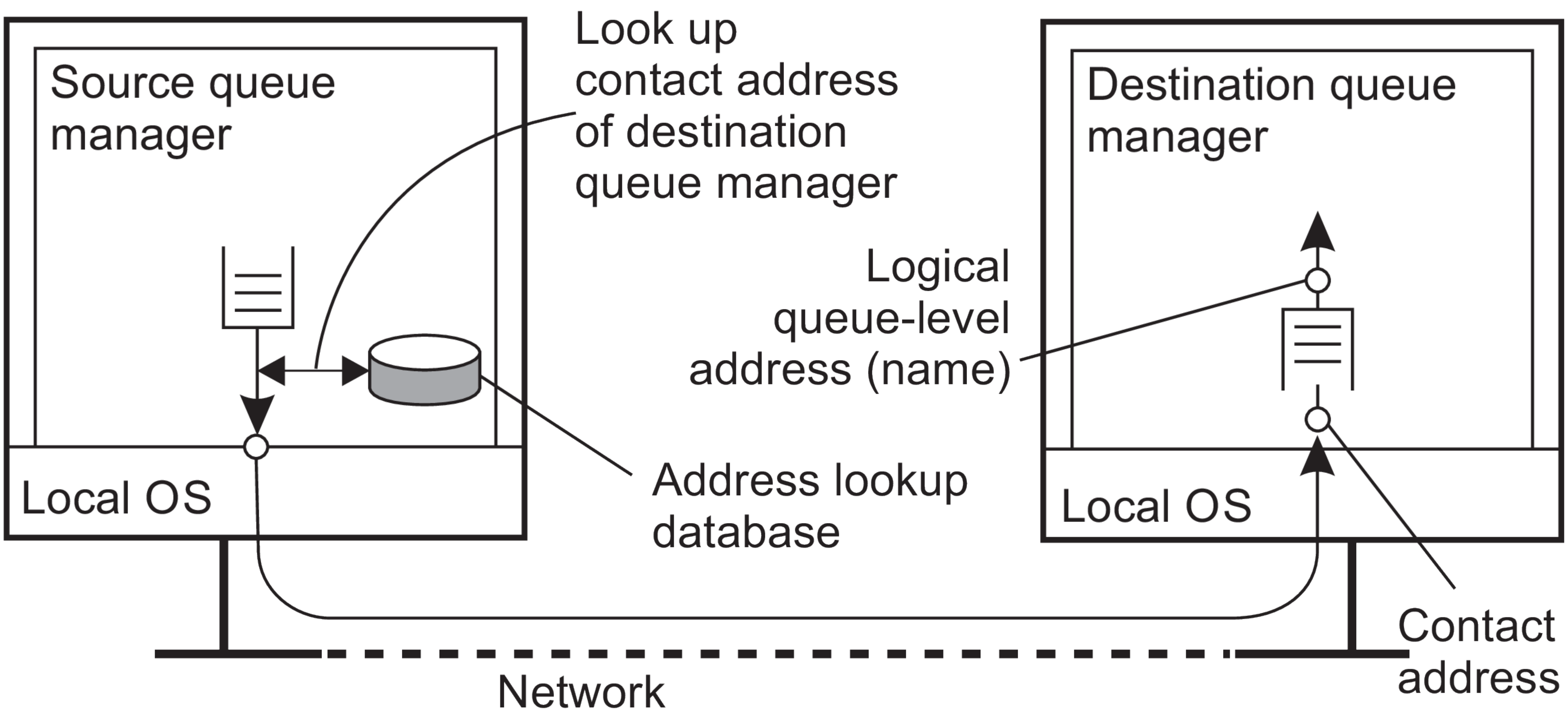
Message Oriented Communication

- General architecture of a message-queueing system
  - Queue managers
    - Applications put messages into local queues and consume messages from local queues
    - Queue managers make sure messages get delivered

Message Oriented Communication

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

Message Oriented Communication



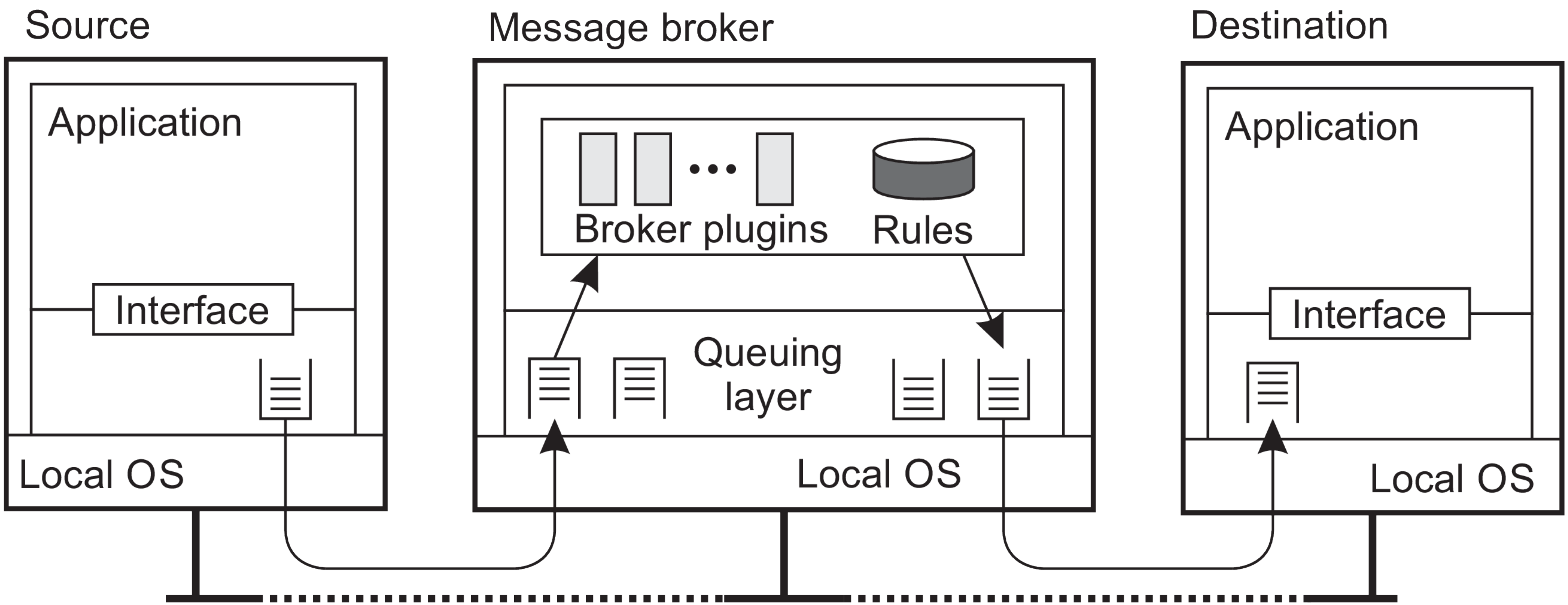
Message Oriented Communication

- Contact addresses
  - (Host, Port)-pair, Protocol (tcp/udp)
- Use special queue managers as routers

Message Oriented Communication

- Observation
  - Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)
- Broker handles 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 (i.e., publish-subscribe capabilities)

Message Oriented Communication



Message Oriented Communication

- Message brokers are built on top of a message-queueing system

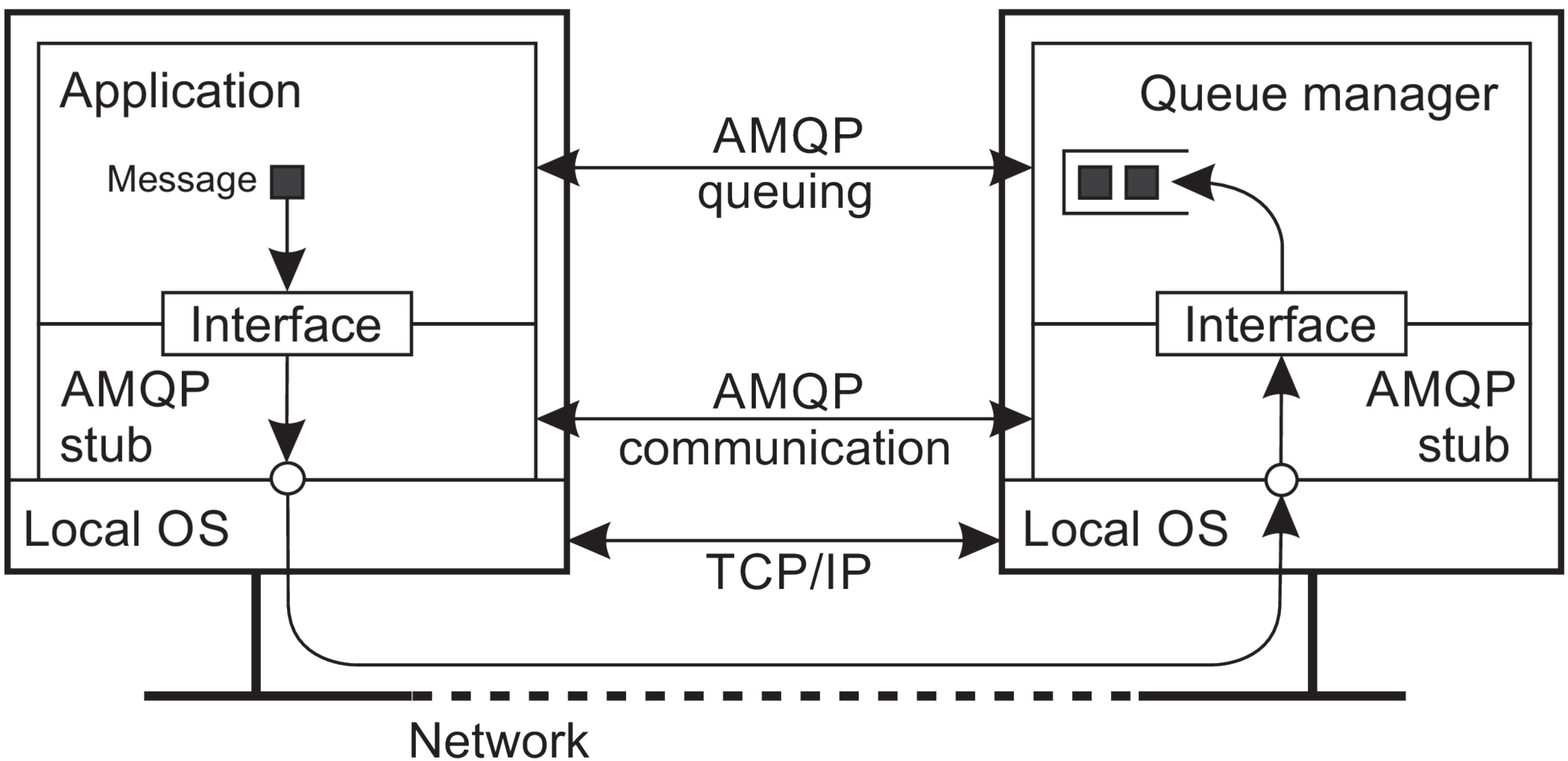
Message Oriented Communication

- Message brokers can be used for
  - **Enterprise Application Integration (EAI)**

Message Oriented Communication

- Example: **Advanced Message-Queueing Protocol (AMQP)**
  - Distinguishing: Messaging service, Messaging protocol, Messaging interface
- Advanced Message-Queuing Protocol was intended to play the same role as, for example, TCP in networks: a protocol for high-level messaging with different implementations.

Message Oriented Communication



Message Oriented Communication

- Basic model
  - Client sets up a (stable) connection, which is a container for several (possibly ephemeral) one-way channels. Two one-way channels can form a session. A link is akin to a socket, and maintains state about message transfers.
-

Message Oriented Communication

- AMPQ communication
  - Application sets up a connection to a queue manager
  - Each connection has several one-way channels
  - Sessions establish bidirectional communication
  - Links transfer mesages

Message Oriented Communication

1. At the sender's side, the message is assigned a unique identifier and is recorded locally to be in an unsettled state. The stub subsequently transfers the message to the server, where the AMQP stub also records it as being in an unsettled state. At that point, the server-side stub passes it to the queue manager.

2. The receiving application (in this case the queue manager), is assumed to handle the message and normally reports back to its stub that it is finished. The stub passes this information to the original sender, at which point the message at the original sender's AMQP stub enters a settled state.

3. The AMQP stub of the original sender now tells the stub of the original receiver that message transfer has been settled (meaning that the original sender will forget about the message from now on). The receiver's stub can now also discard anything about the message, formally recording it as being settled as well.

Message Oriented Communication

- AMPQ messaging
    - Happens in layer above communication layer
    - Takes place between nodes: producer, consumer, or queue

Message Oriented Communication

```python
import rabbitpy

def producer():
  connection = rabbitpy.Connection() # Connect to RabbitMQ server
  channel = connection.channel()     # Create new channel on the connection

  exchange = rabbitpy.Exchange(channel, ’exchange’) # Create an exchange
  exchange.declare()

  queue1 = rabbitpy.Queue(channel, ’example1’) # Create 1st queue
  queue1.declare()

  queue2 = rabbitpy.Queue(channel, ’example2’) # Create 2nd queue
  queue2.declare()

  queue1.bind(exchange, ’example-key’) # Bind queue1 to a single key
  queue2.bind(exchange, ’example-key’) # Bind queue2 to the same key

  message = rabbitpy.Message(channel, ’Test message’)
  message.publish(exchange, ’example-key’) # Publish the message using the key
  exchange.delete()
```

Message Oriented Communication

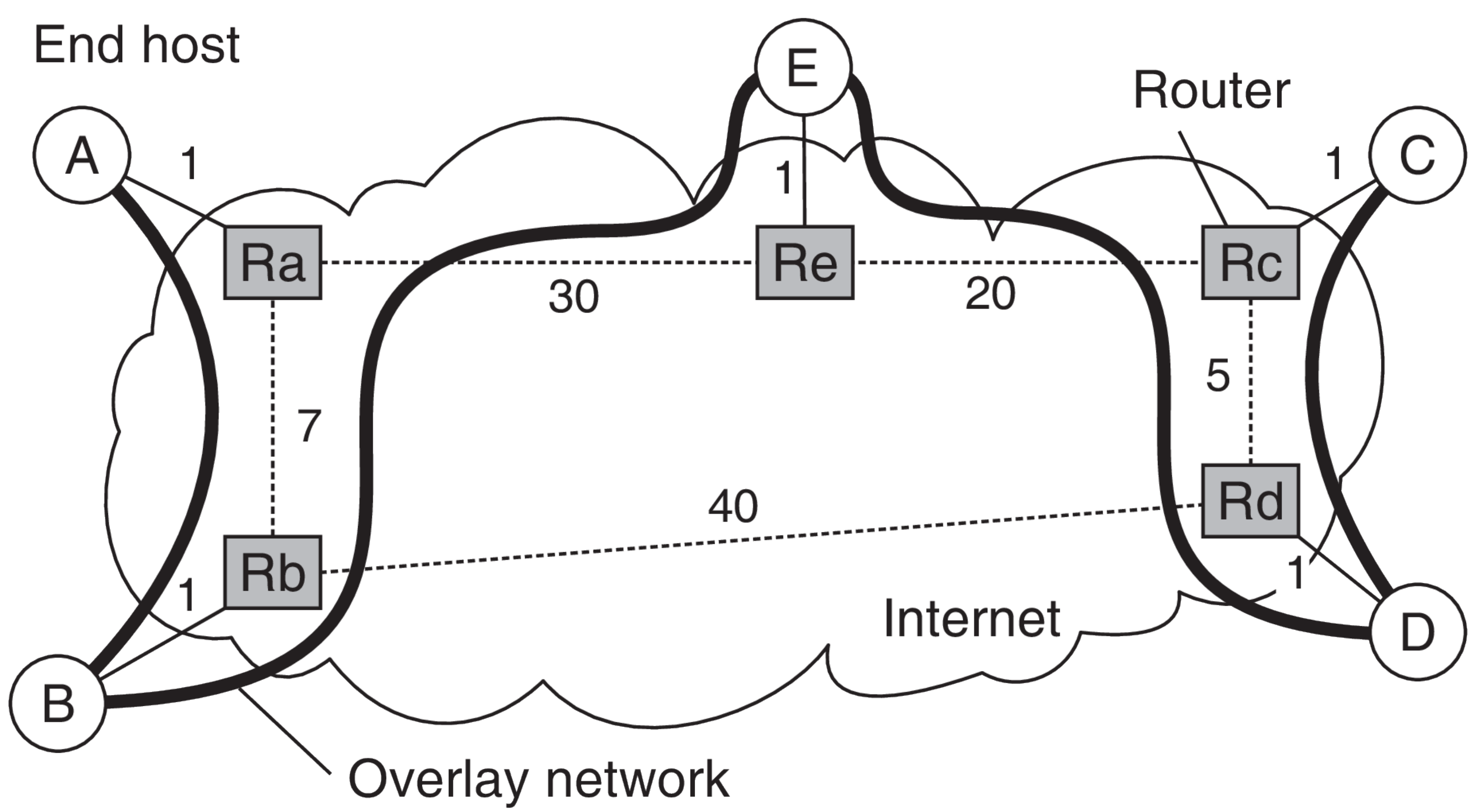
```python
import rabbitpy

def consumer():
  connection = rabbitpy.Connection()
  channel = connection.channel()

  queue = rabbitpy.Queue(channel, 'example1')

  # While there are messages in the queue, fetch them using Basic.Get
  while len(queue) > 0:
    message = queue.get()
    print('Message Q1: %s' % message.body.decode())
    message.ack()

  queue = rabbitpy.Queue(channel, 'example2')

  while len(queue) > 0:
    message = queue.get()
    print('Message Q2: %s' % message.body.decode())
    message.ack()
```

Message Oriented Communication

- Multicasting
  - Application-level tree-based multi-casting
    - Organize nodes of a distributed system into an overlay network and use that network to disseminate data:
    - Oftentimes a tree, leading to unique paths
    - Alternatively, also mesh networks, requiring a form of routing

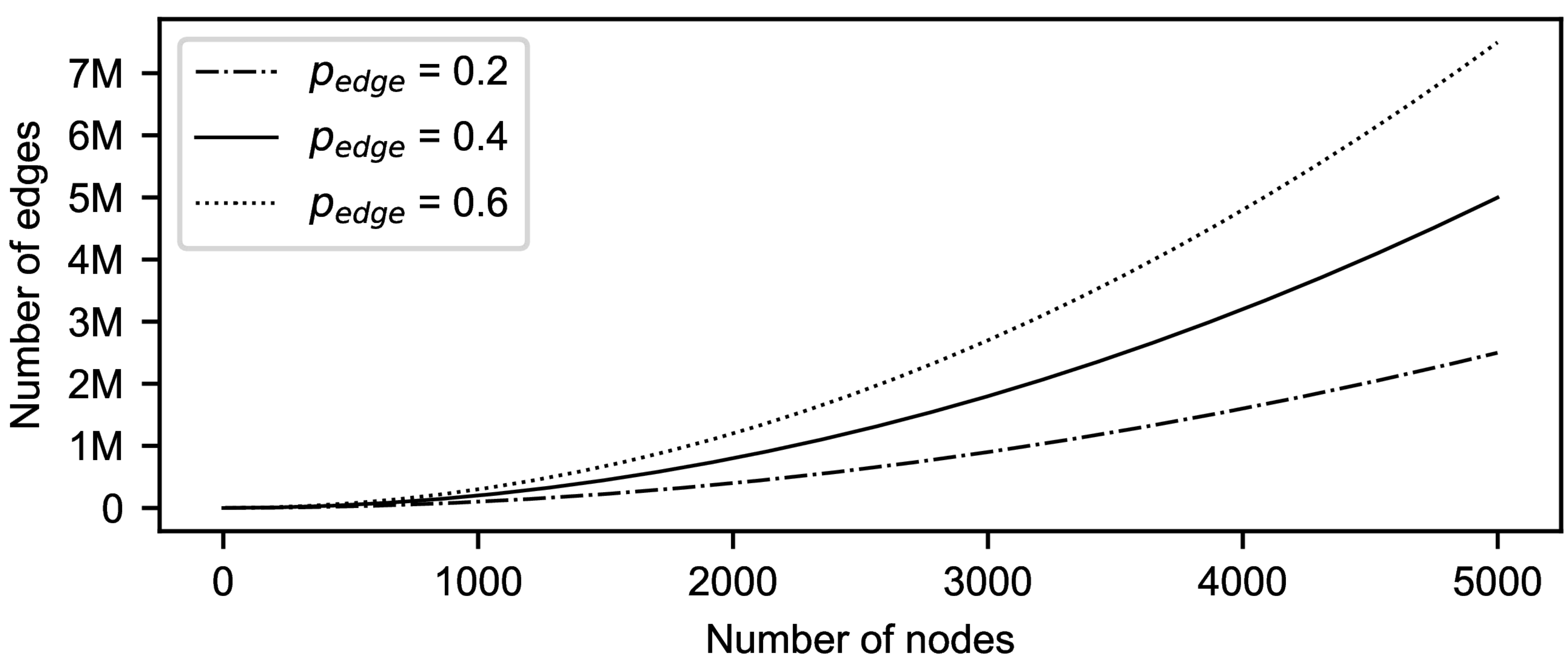
Message Oriented Communication

- Overlay networks allow multi-casting
  - Link stress: How often does a packet cross the same link
  - Stretch: delay in overlay / delay in network



Message Oriented Communication

- Flooding-based multicasting
  - P simply sends a message m to each of its neighbors. Each neighbor will forward that message, except to P, and only if it had not seen m before.



Message Oriented Communication

- Gossip-based data dissemination
  - Epidemic protocols
  - Assume there are no write–write conflicts
  - Update operations are performed at a single server
  - A replica passes updated state to only a few neighbors
  - Update propagation is lazy, i.e., not immediate
  - Eventually, each update should reach every replica

Message Oriented Communication

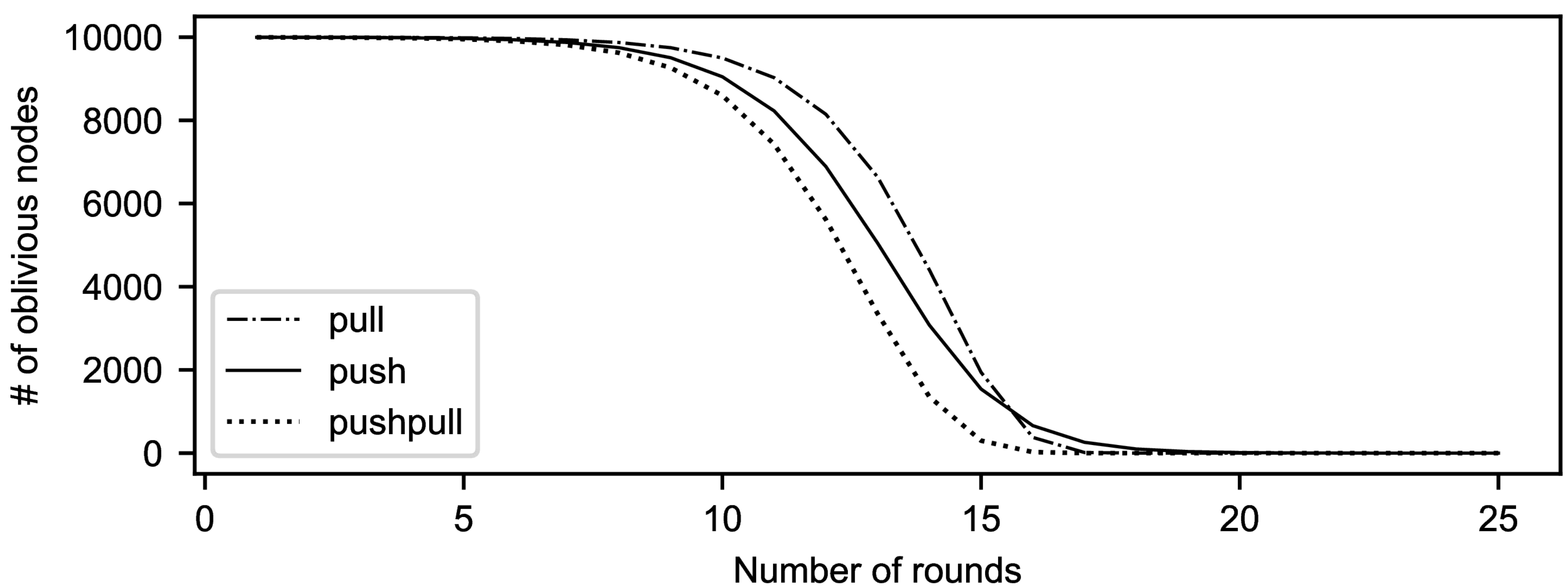
- Two forms of epidemics
  - Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
  - Rumor spreading: A replica which has just been updated (i.e., has been contaminated), tells several other replicas about its update (contaminating them as well).

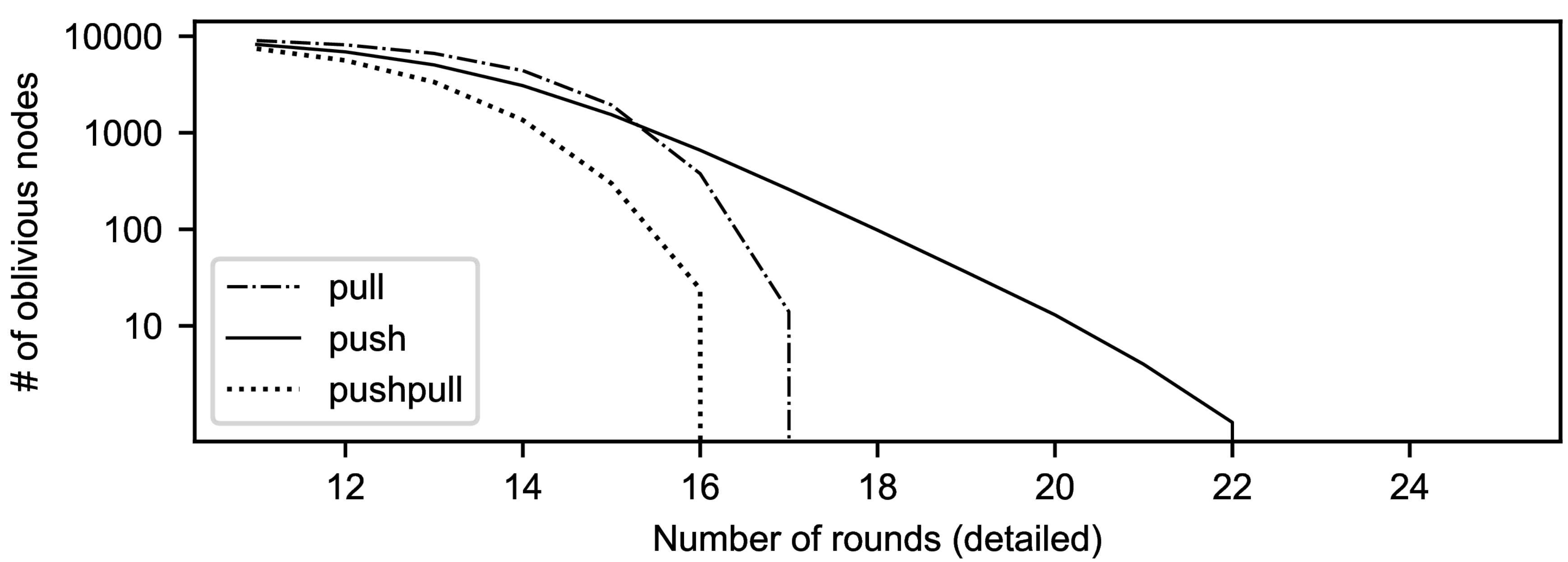
Message Oriented Communication

- Anti-entropy
  - Principle operations
  - A node P selects another node Q from the system at random.
    - Pull: P only pulls in new updates from Q
    - Push: P only pushes its own updates to Q
    - Push-pull: P and Q send updates to each other

- Observation
- For push-pull it takes O(log(N)) rounds to disseminate updates to all N nodes (round = when every node has taken the initiative to start an exchange).

Message Oriented Communication

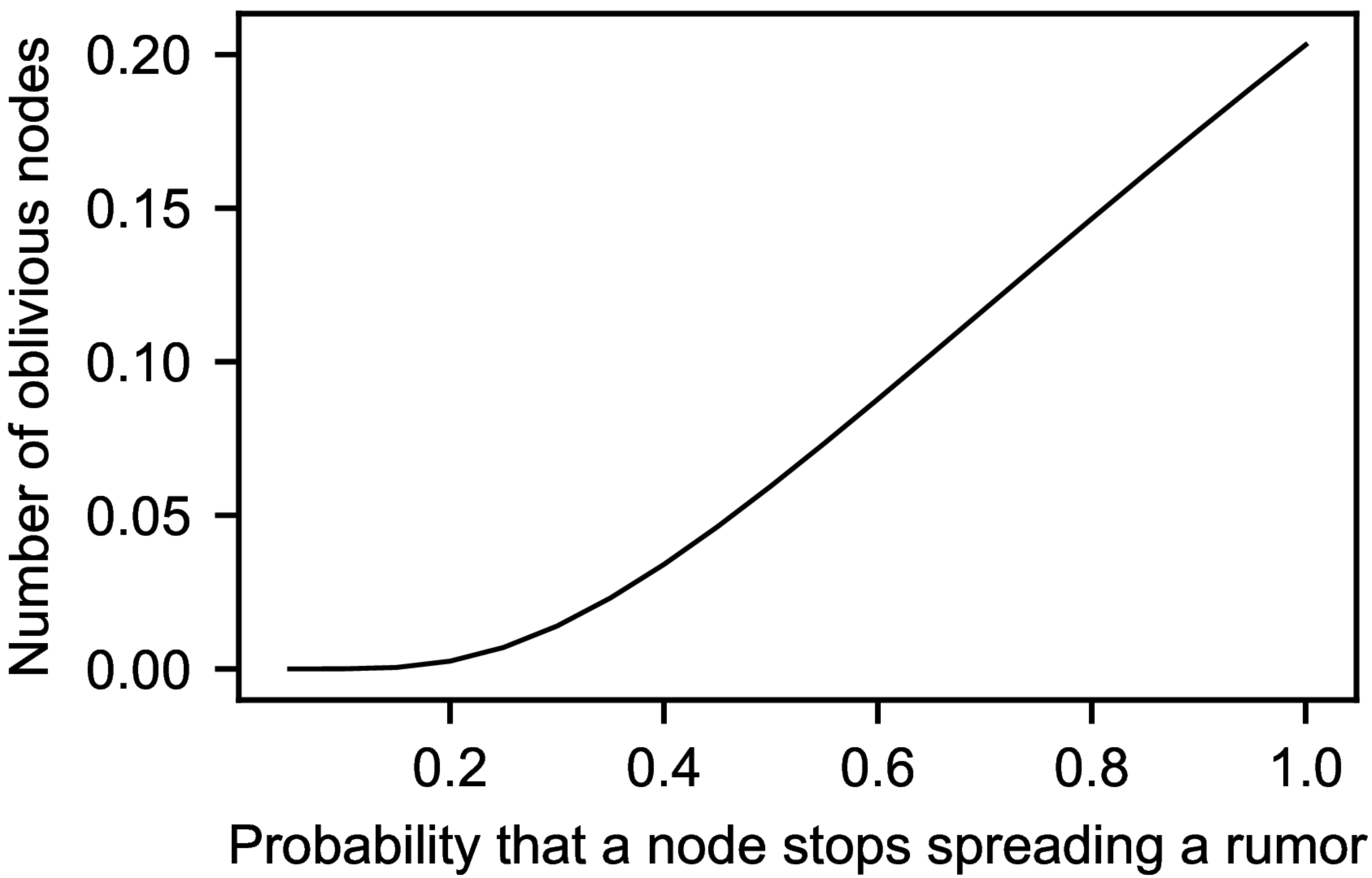




Message Oriented Communication

- Rumor spreading:
  - Basic model
  - A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability pstop.
  - Observation
  - If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers
  - $s = e^{-(1/p_{\text{stop}}+1)(1-s)}$

Message Oriented Communication



Message Oriented Communication

- Removing data:
  - Make a deletion into an update to a NULL content
    - Sending out death certificates
      - Should become dormant

Stream Oriented Communication

- Streams:
  - Timing is crucial
- Continuous media
  - Meaning of message depends on temporal relationship to previous messages
    - Motion , Audio
- Discrete media
  - text, still images

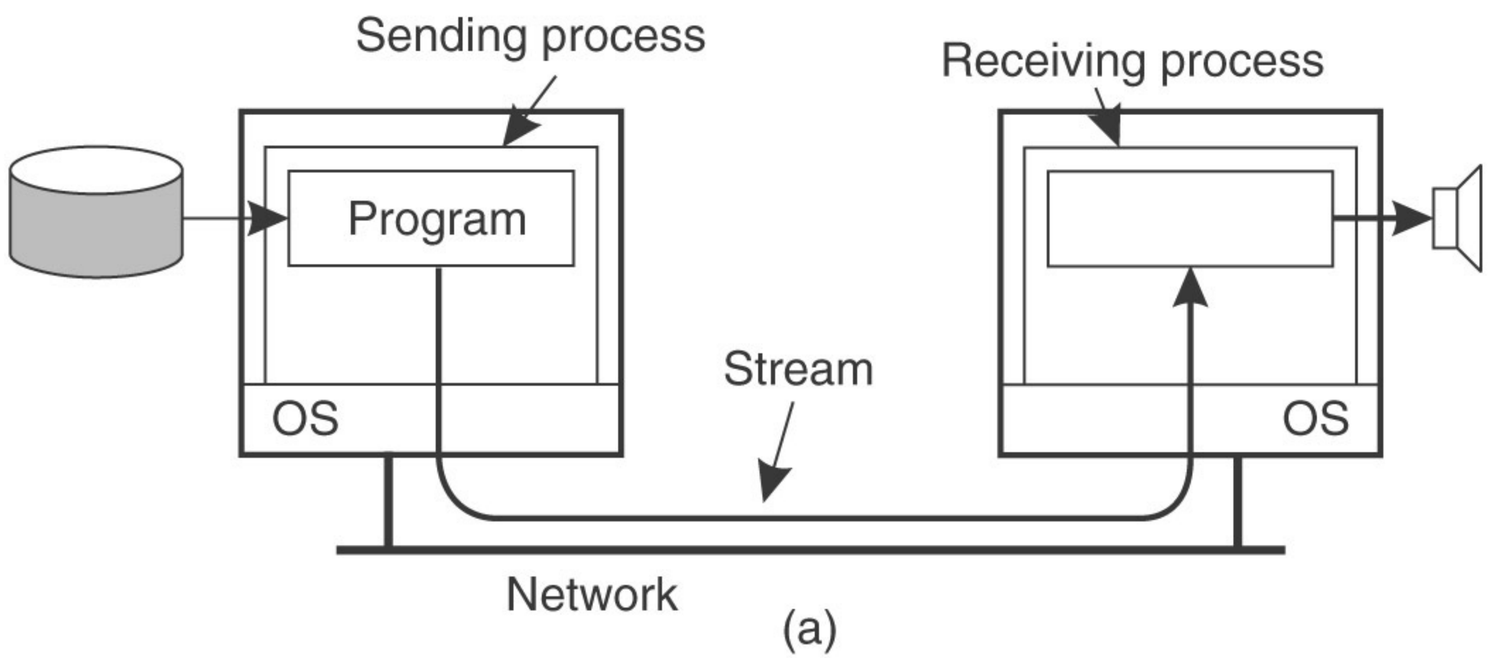
Stream Oriented Communication

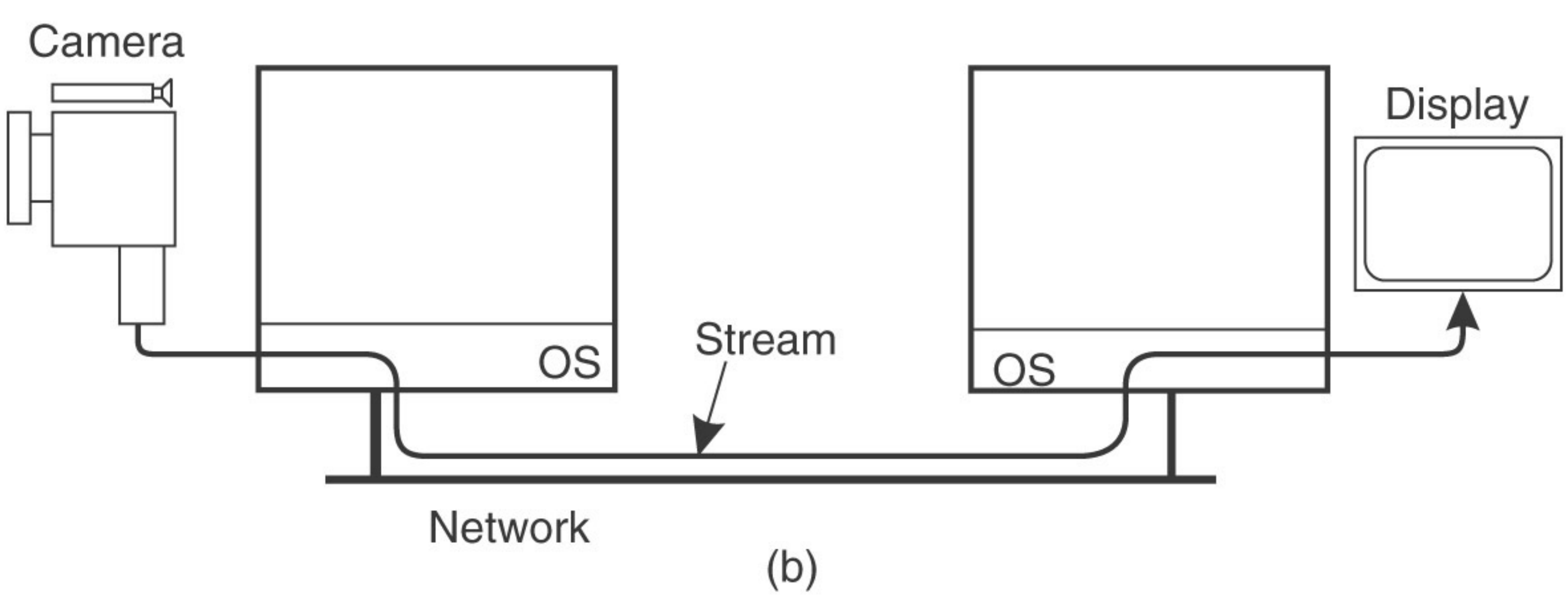
- Data stream: sequence of data units
  - Asynchronous transmission mode:
    - Data items are transmitted in order
    - without timing constraints
  - Synchronous transmission mode:
    - Data items are transmitted in order
    - Maximum end-to-end delay for each unit in the stream
  - Isochronous transmission mode:
    - Data units are transferred on time
    - Maximum and minimum end-to-end delay
      - “Bounded delay jitter"

Stream Oriented Communication

- Simple streams
- Complex streams
  - consist of several related simple streams, the sub-streams

Stream Oriented Communication



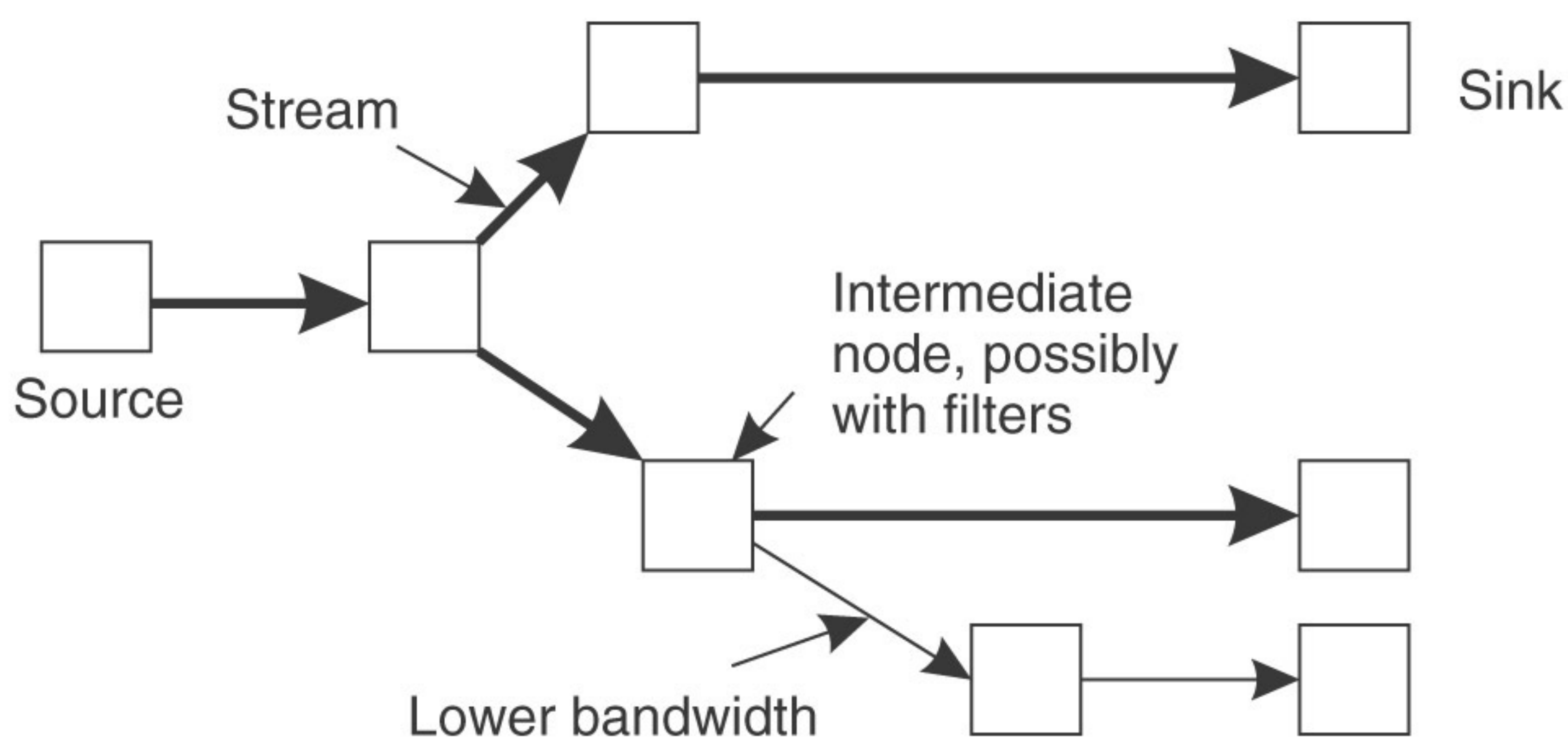


(a) stream between two processes

(b) stream between two devices

Stream Oriented Communication

- Multicasting
  - Receivers can have different requirements
  - Use *filters* to adjust quality

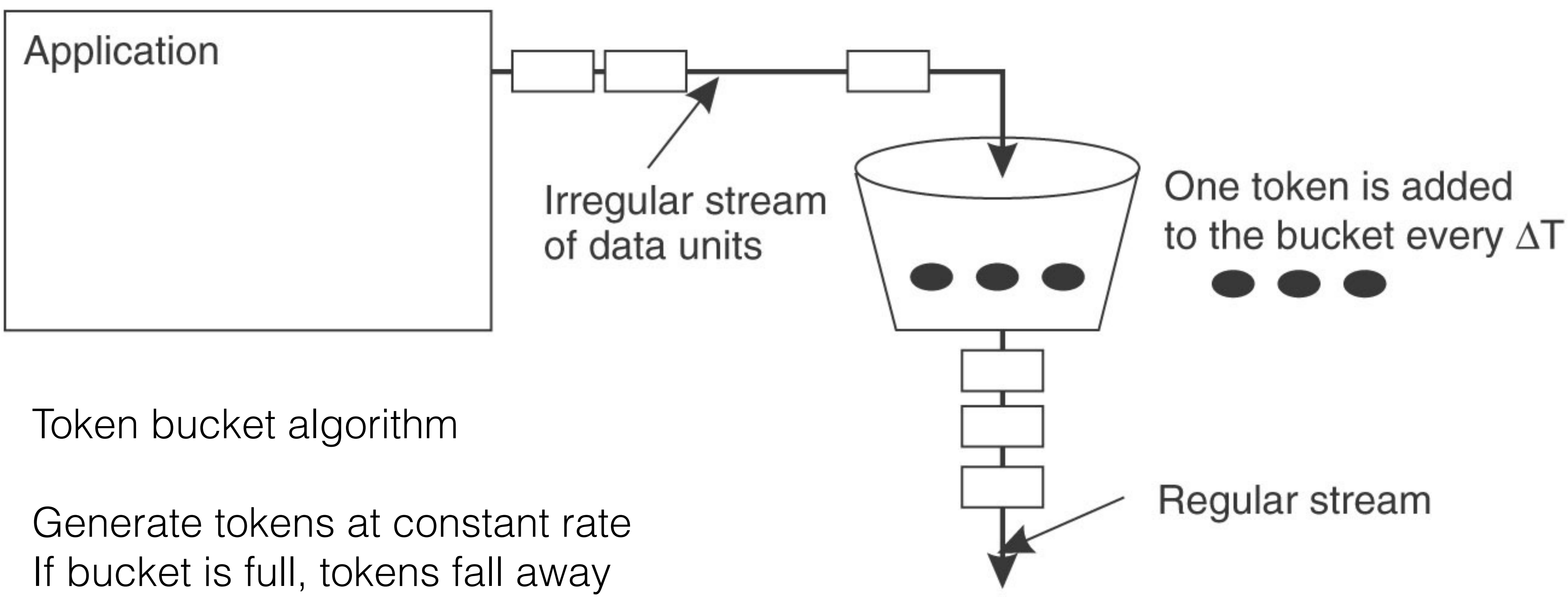


Stream Oriented Communication

- Quality of Service (QoS)
  - Flow specification: bandwidth requirements, transmission rates, delays, …

Characteristics of the Input	Service Required
Maximum data unit size (bytes)	Loss sensitivity (bytes)
Token bucket rate (bytes/sec)	Loss interval (μsec)
Token bucket size (bytes)	Burst loss sensitivity (data units)
Maximum transmission rate (bytes/sec)	Minimum delay noticed (μsec)
	Maximum delay variation (μsec)
	Quality of guarantee

Stream Oriented Communication



Token bucket algorithm

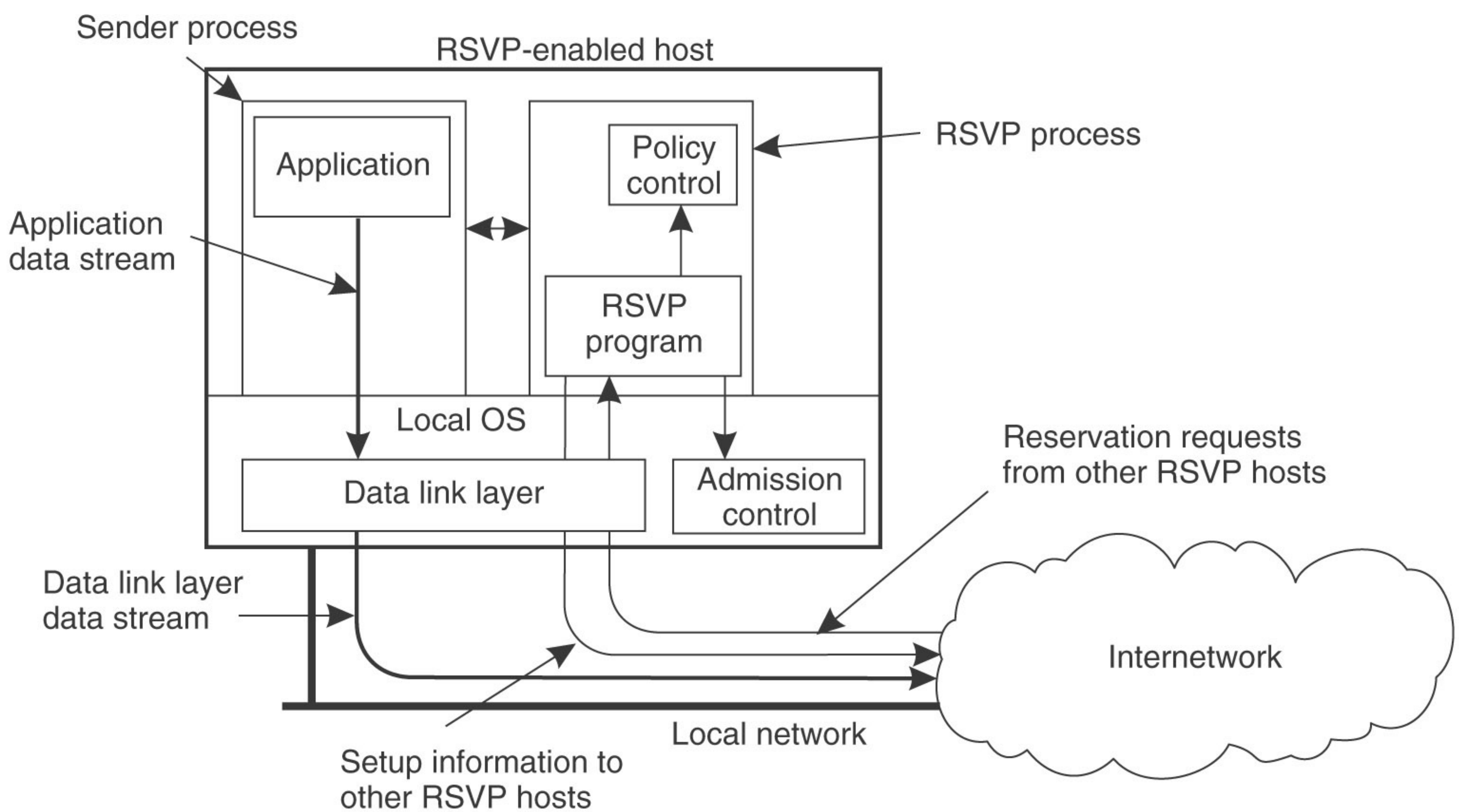
Generate tokens at constant rate
If bucket is full, tokens fall away
Each time application sends data,
needs to remove tokens from bucket

Stream Oriented Communication

- Currently, no model for
  - specifying QoS parameters
  - describing resources in a communication system
  - translating QoS parameters to resource usage

Stream Oriented Communication

- QoS protocol: Resource reSerVation Protocol (RSVP)



Stream Oriented Communication

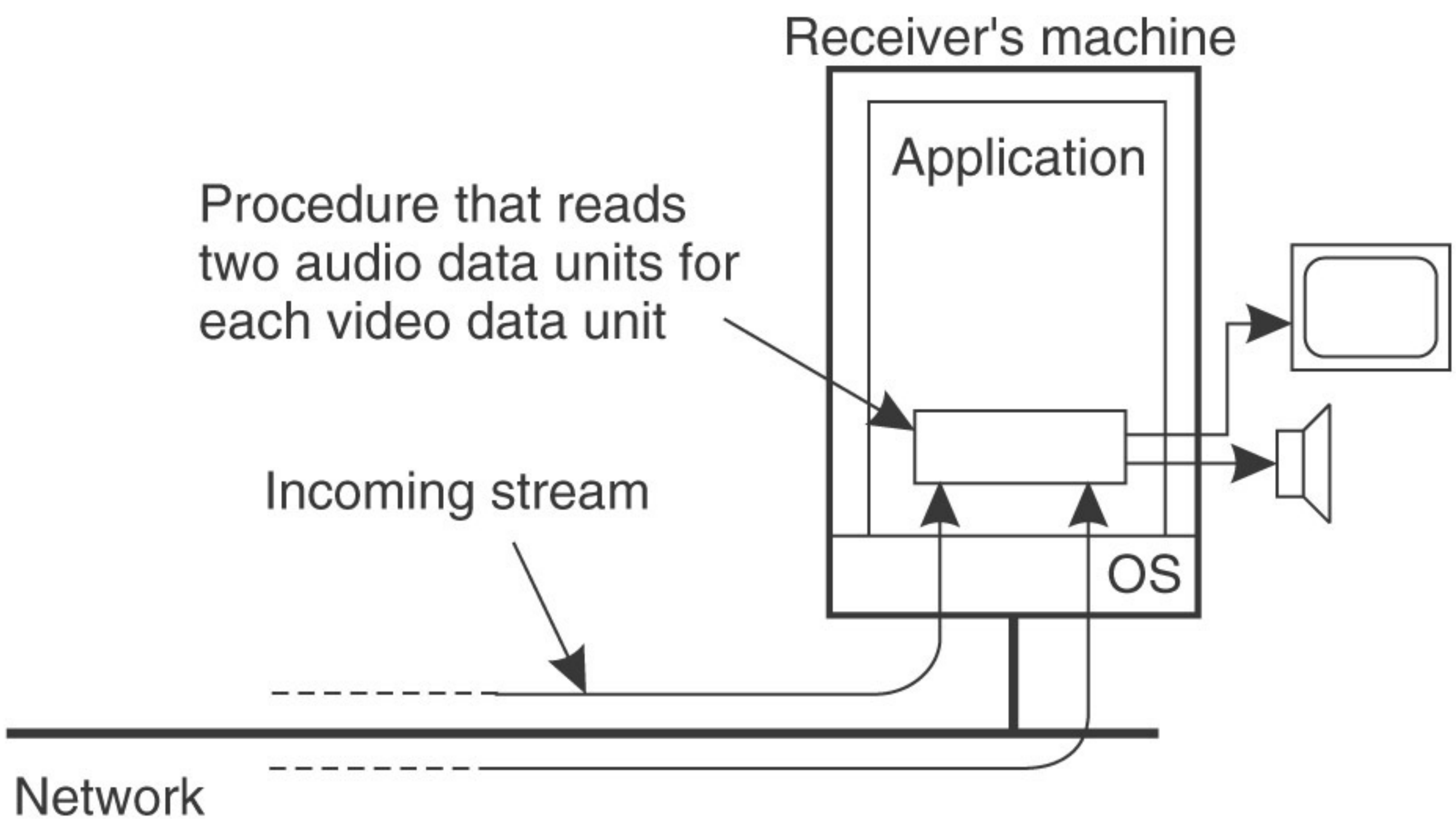
- RSVP
  - Senders provide flow specification
  - Hand it over to RSVP process
  - RSVP process stores specification
  - Sender sets up path to receiver(s)
    - providing flow specification to all intermediate nodes
  - RSVP server when receiving a reservation request:
    - Checks whether enough resources are available
    - Checks whether receiver has permission to make the reservation

Stream Oriented Communication

- Stream synchronization
  - Sub streams in a complex stream need to be synchronized
  - Simple form: discrete data stream (slides) and continuous data stream (audio)
  - Complex form: Synchronizing video and audio stream, two audio streams for stereo (with max. jitter of less than 20 μsec
- Need to synchronize between data units

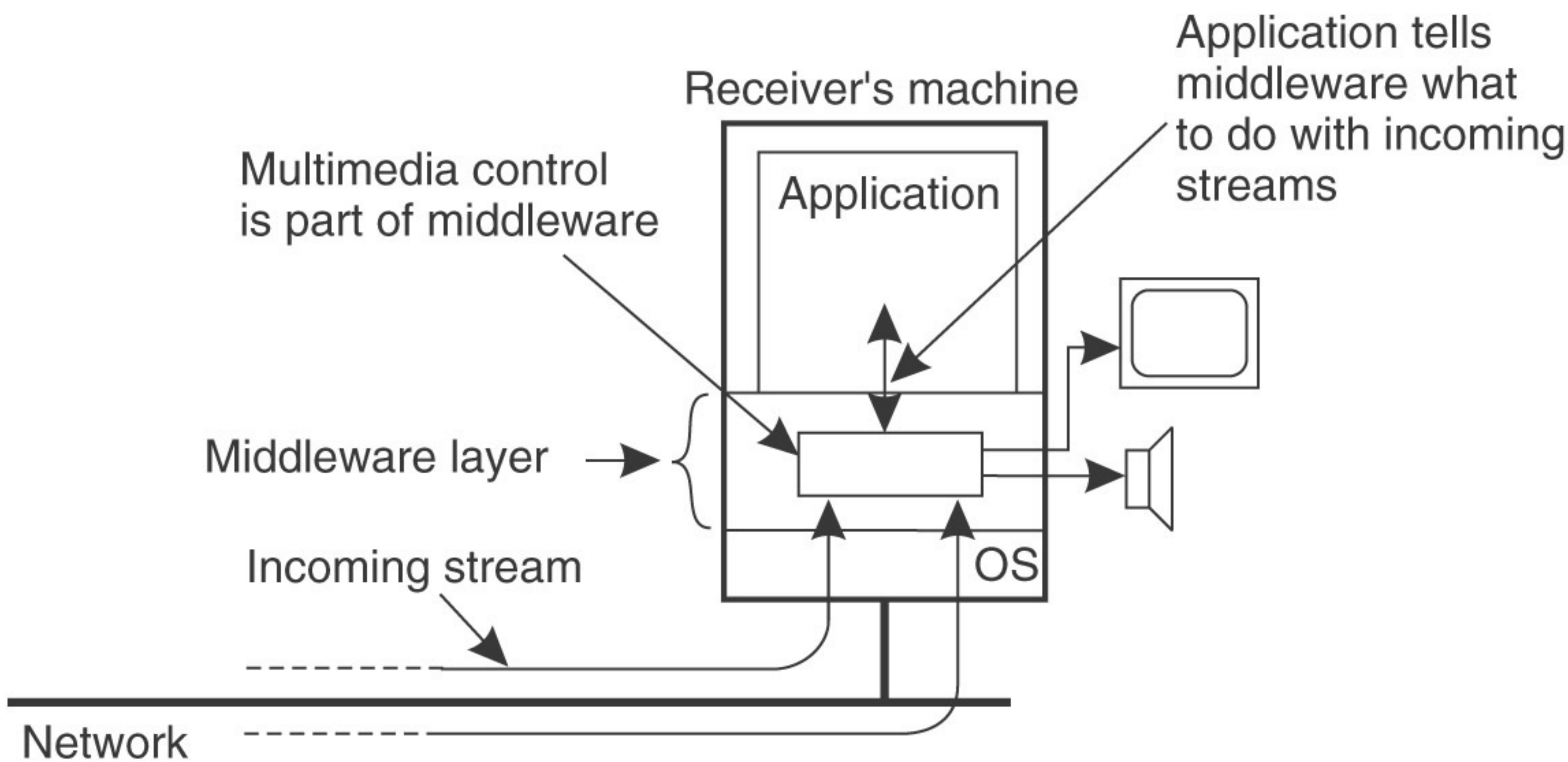
Stream Oriented Communication

- Explicit synchronization at the data level



Stream Oriented Communication

- Synchronization at high level



Stream Oriented Communication

- Synchronization at high level:
  - Multimedia middleware offers interfaces for controlling video and audio streams
- Multiplex different streams into a single stream:
  - MPEG streams