Higher-Level Communication

Communication techniques

- TCP/IP protocol stack
- Sockets: OS service for using network communication
  - Low-level
- Basic server model:
  - A process implementing a specific service on behalf of collection of clients.
  - Waits for an incoming request from a client, and ensures that it is taken care of.
  - Waits for another incoming request after responding
- Concurrent servers: Use a dispatcher, which passes the request to a separate process/thread

- RPC’s : Remote Procedure Calls
  - Higher-level than sockets

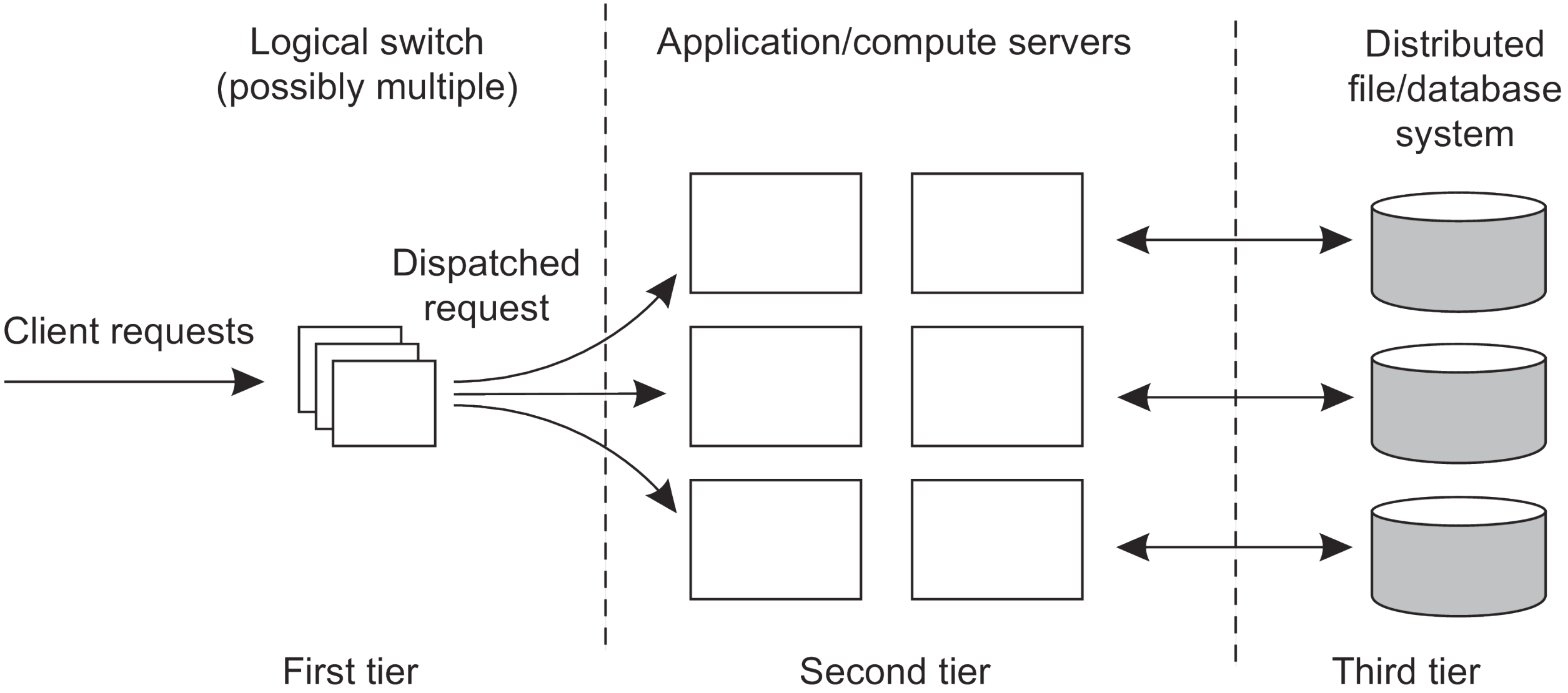
Client/Server

- Stateless servers are a viable design option
- Dont rely on accurate information about the status of a client after having handled a request
  - Don’t record whether a file has been opened (simply close it again after access)
  - Don’t promise to invalidate a client’s cache
  - Don’t keep track of clients
- Many consequences:
  - Clients and servers are completely independent
  - State inconsistencies due to client or server crashes are reduced
  - Possible loss of performance if server cannot anticipate client behavior (such as prefetching data)

Decoupled Communication

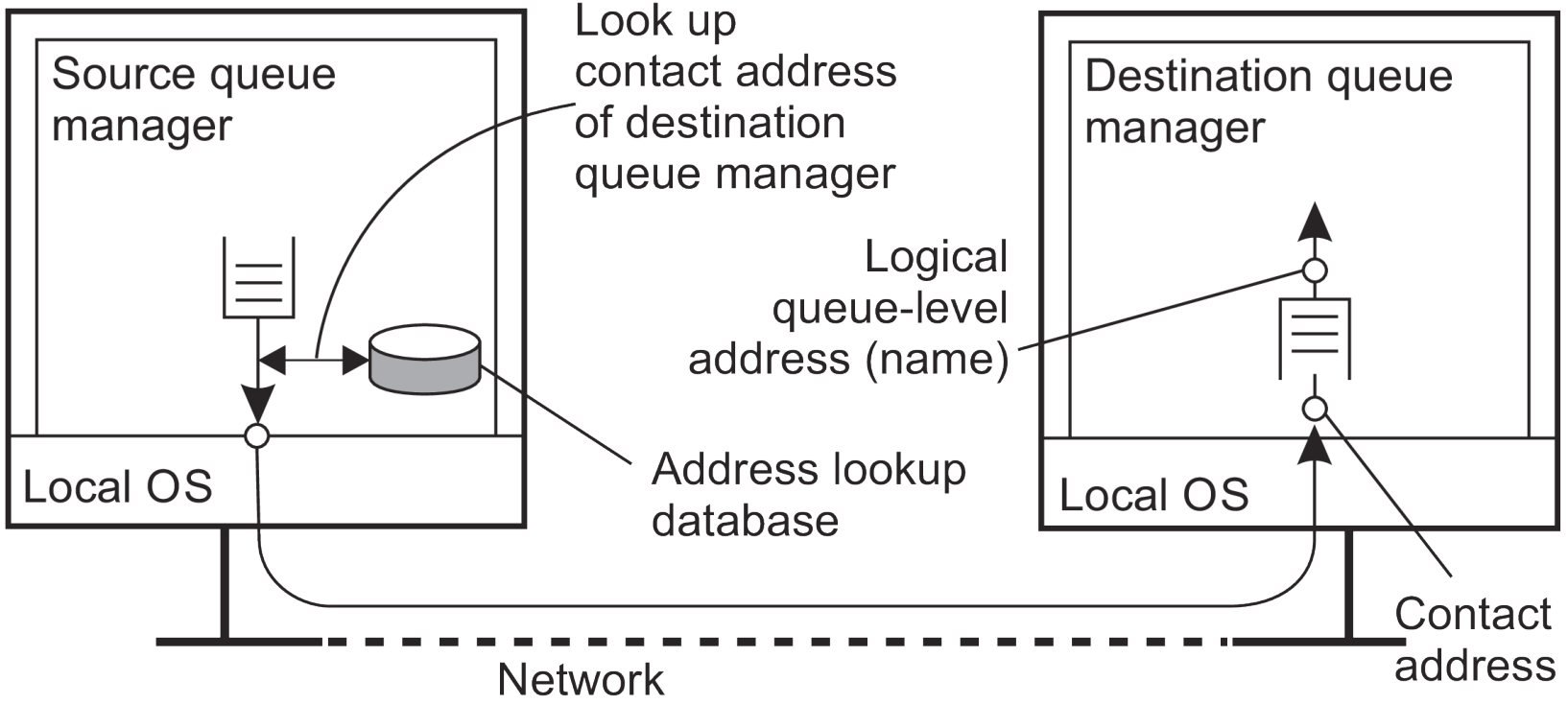
- **Space decoupling**. Interacting parties don’t need to know about each other
- **Time decoupling**. Need not be actively participating/running at the same time.
- **Synchronization decoupling**. Sender, receiver should not block
  - Production/consumption do not happen in main flow of control
- Increases scaling, since there is no explicit dependency

Multi-tier Server Architecture



Higher Level Messaging: Message Queues

- Easy to make programming mistakes with sockets
- Popular patterns of usage that can be re-used (like client/server)
- Message Queueing systems provide a higher level of expression

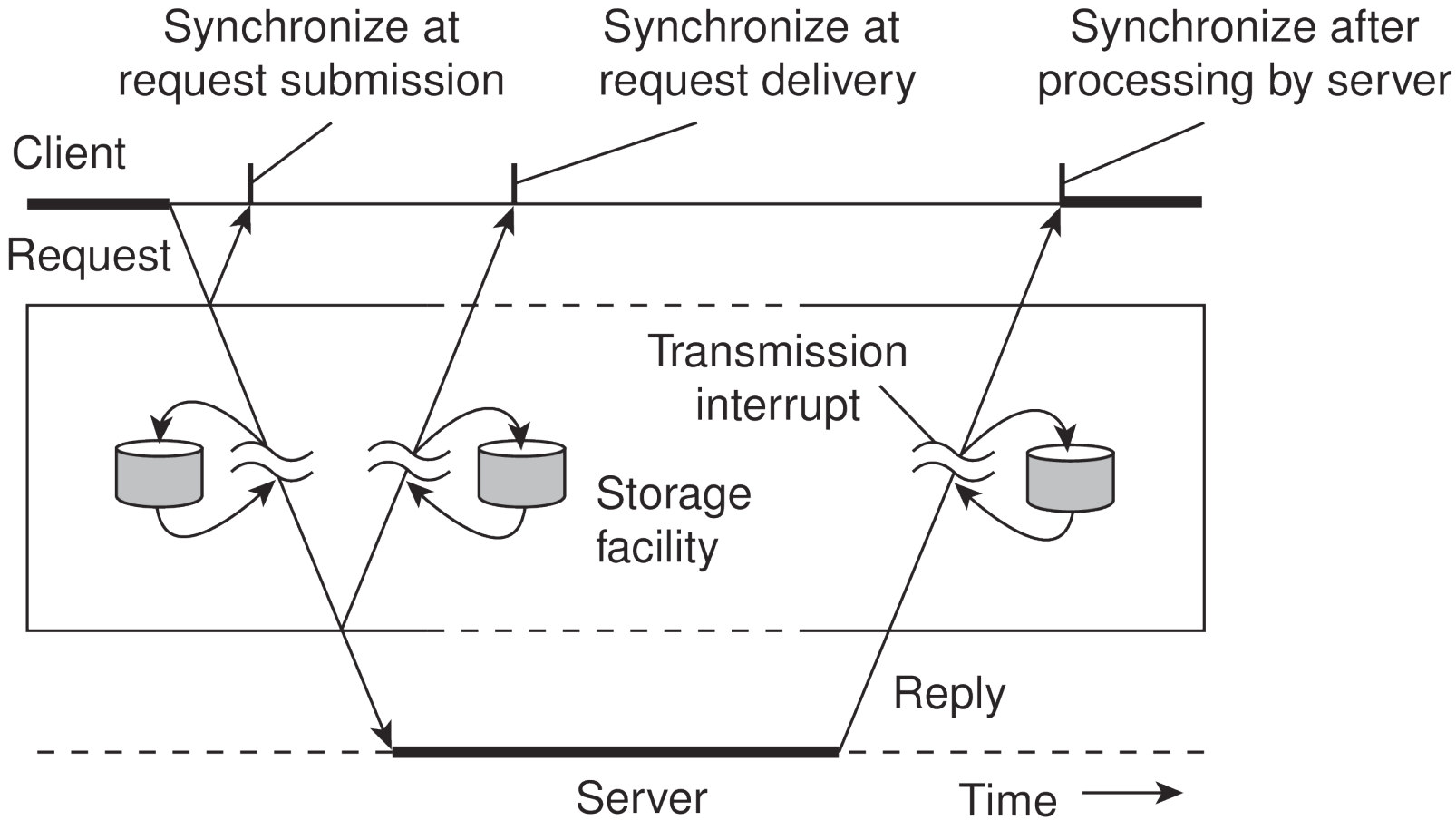


Message Queue Challenges

- How do we handle dynamic components, i.e., pieces that go away temporarily? Do we formally split components into "clients" and "servers" and mandate that servers cannot disappear? What then if we want to connect servers to servers? Do we try to reconnect every few seconds?
- How do we represent a message on the wire? How do we frame data so it's easy to write and read, safe from buffer overflows, efficient for small messages, yet adequate for the very largest videos of dancing cats wearing party hats?
- How do we handle messages that we can't deliver immediately? Particularly, if we're waiting for a component to come back online? Do we discard messages, put them into a database, or into a memory queue?
- How do we handle I/O? Does our application block, or do we handle I/O in the background? This is a key design decision. Blocking I/O creates architectures that do not scale well. But background I/O can be very hard to do right.

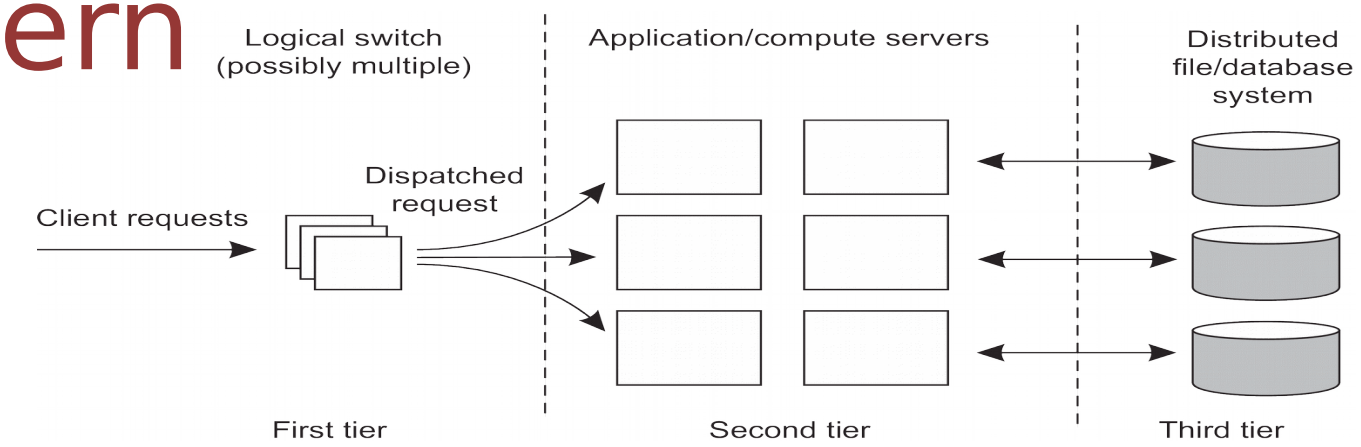
Persistent Communication

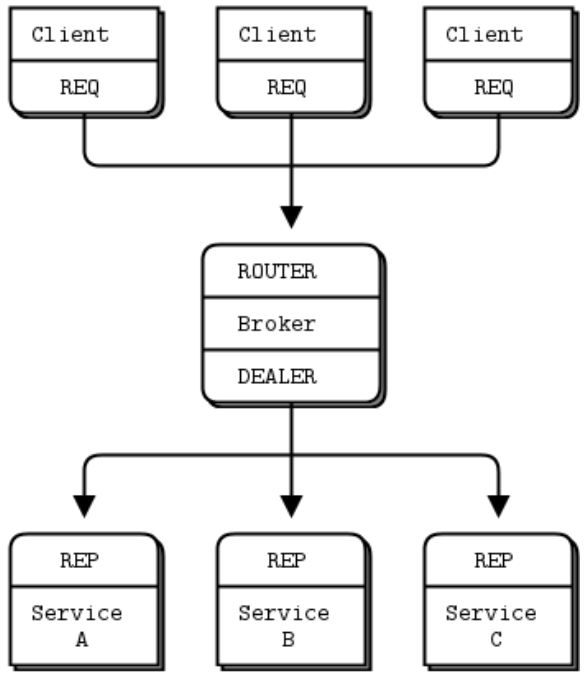
- Socket based: transient communication.
  - Messages sent/rcvd are ephemeral
- Persistent communication: messages are stored by messaging layer



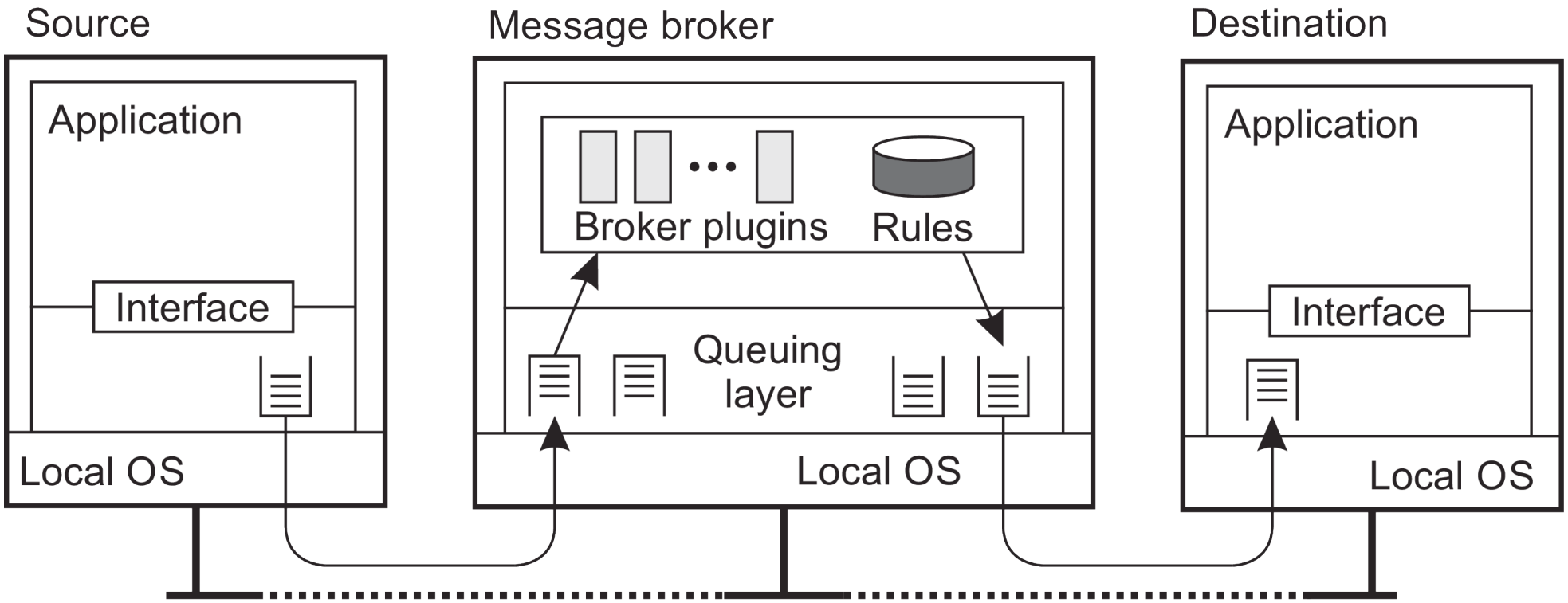
Broker Design Pattern

- Add a message broker between client and server
- Easier to scale client/server
- Clients decoupled from servers
- Only broker is static





Message Broker: General Architecture



Common message brokers: RabbitMQ, Apache Kafka, ...

Request-Reply with ZeroMQ

- Higher-level socket-like interface
- No broker required

Server

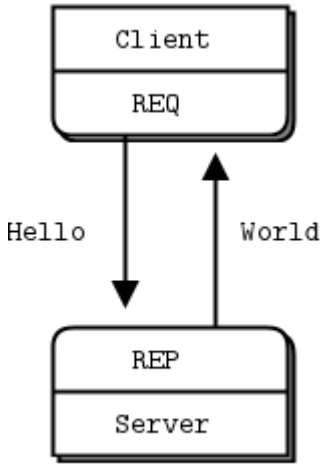
```
1  import zmq
2  context = zmq.Context()
3
4   p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
5   p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
6  s = context.socket(zmq.REP)    # create reply
7                                 socket
8   s.bind(p1)                        # bind socket to
9   s.bind(p2)                        address   # bind
10 while True:                        socket to address
11   message = s.recv()                # wait for incoming
12   if not "STOP" in message:         message   # if not to
13   s.send(message + "*")             stop...
14   else:                             # append "*" to message
15   break                              # else...
                                       # break out of loop and
                                       end
```

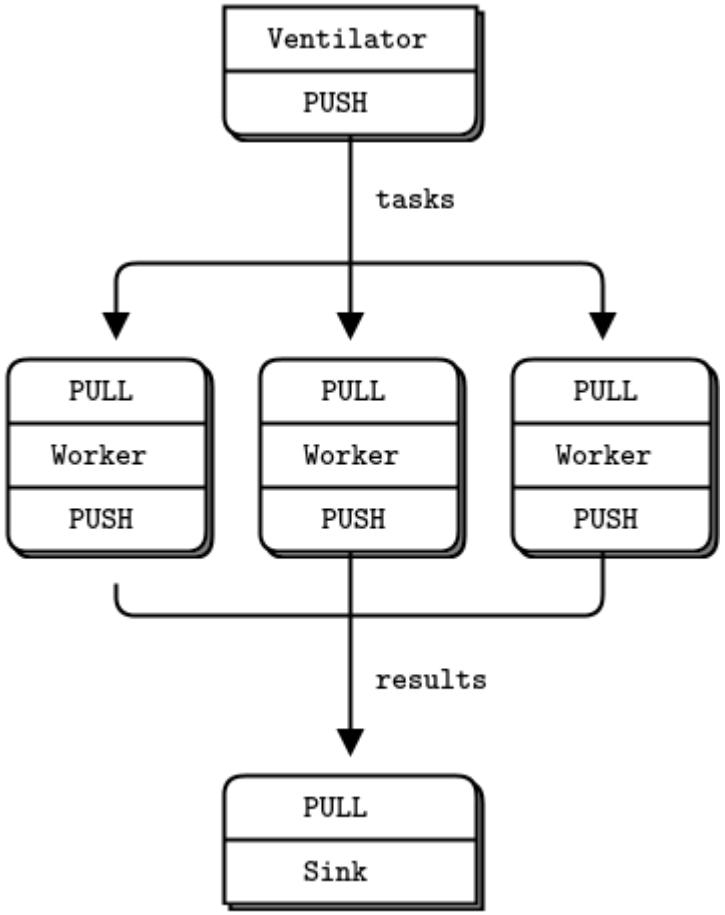
Request-Reply

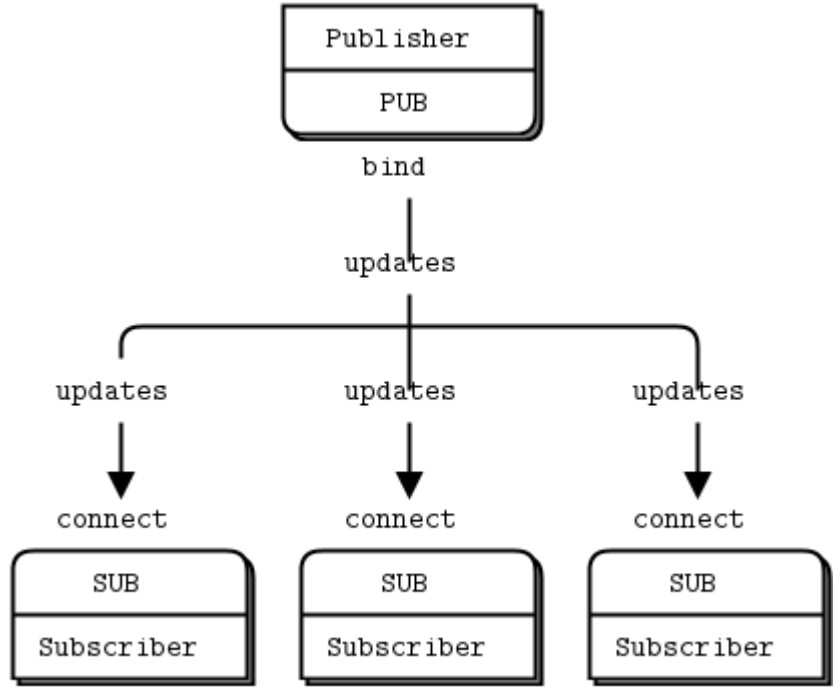
Client

```
1 import zmq
2 context = zmq.Context()
3
4 php = "tcp://"+ HOST +":"+ PORT # how and where to
connect
5 s = context.socket(zmq.REQ) # create socket
6                               # block until
7 s.connect(php)                connected   # send
8 s.send("Hello World")         message
9 message = s.recv()            # block until
10 s.send("STOP")               response   # tell
11 print message                 server to stop  #
                                print result
```

Queue-based communication patterns







Publish-Subscribe

- Need flexible communication models and systems
- Dynamic and decoupled nature of applications
- Point-to-point and synchronous communication is not ideal
  - Leads to rigid and static applications
  - Elastic applications essential in large clouds and data centers (failures, pricing, etc.)
- Publish-Subscribe is one interaction scheme for addressing these problems
- Enables loosely coupled systems
- Fully decouple time, space, and synchronization

Publish-Subscribe basics

- Publishers: Generate events of different types
- Subscribers: Express interest in an event or pattern of events
- Get notified if events match their registered interest
- Producers publish info on a “software bus”
- Usually requires an event-service
  - Provides storage and management for subscription
  - What type of events exist, where to deliver, etc.
  - Also efficient delivery of messages

More Publish-Subscribe

- Subscribers register interest by calling subscribe() on event service, without knowing the sources of the event
- Events pushed via publish()
- Publishers can also advertise the kind of events they will generate in future

Decoupled Communication

- **Space decoupling**. Interacting parties don’t need to know about each other
  - Publishers/subscribers don’t need to hold references to each other
- **Time decoupling**. Need not be actively participating/running at the same time.
  - Subscribers can get disconnected and be notified later
- **Synchronization decoupling**. Sender, receiver should not block
  - Publishers not blocked while producing events
  - Subscribers can get async notified via callbacks
  - Production/consumption do not happen in main flow of control
- Increases scaling, since there is no explicit dependency

How coupled are other communication patterns?

- Message Passing: Producer and consumer are coupled in time and space
  - Transient communication, not persistent
- RPC
  - Space coupling: invoking object holds remote reference to each invoke
- Async RPC: fire and forget, Futures and Promises based programming
  - Decoupled

Publish-subscribe in ZeroMQ

Server

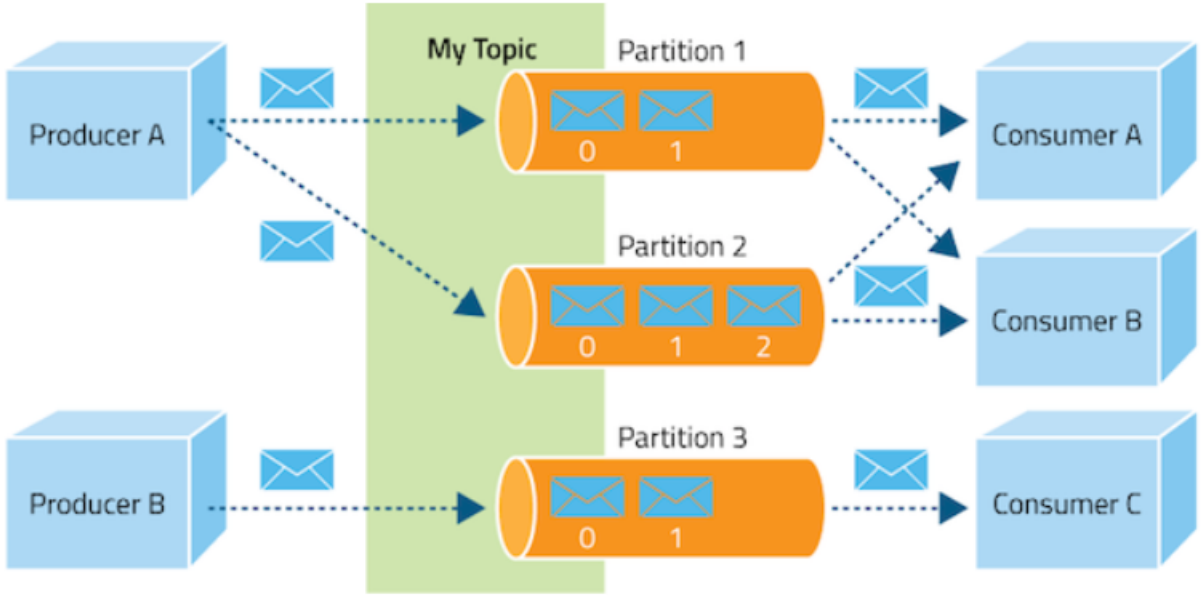
```
1 import zmq, time
2
3 context = zmq.Context()                    # create a publisher socket
4 s = context.socket(zmq.PUB)                # how and where to
5 p = "tcp://"+ HOST +":"+ PORT              communicate   # bind socket
6 s.bind(p)                                  to the address
7 while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

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
6 s.connect(p)                               communicate   # connect to
7 s.setsockopt(zmq.SUBSCRIBE,                the server
"TIME")                                      # subscribe to TIME
8 for i in range(5):  # Five iterations      messages
10   time = s.recv()  # receive a message
11   print time
```

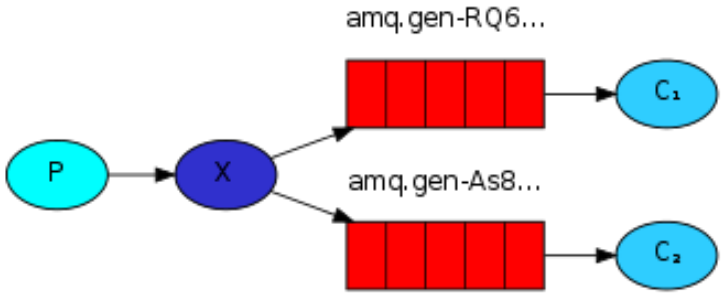
RabbitMQ, Kafka

- Modern pub/sub systems with persistent broker/agent
- Some “broker” should always be running
- Message routing based on key/topic
- Message Queues can also be persistent/durable



RabbitMQ Publish

```
import pika
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='fanout')
message = 'info:Hello'
channel.basic_publish(exchange='logs', routing_key='', body=message)
connection.close()
```



RabbitMQ Subscribe

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='fanout')   #broadcasts to all
result = channel.queue_declare(queue='', exclusive=True)  #close if no consumers left
queue_name = result.method.queue
channel.queue_bind(exchange='logs', queue=queue_name)
print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] %r" % body)

channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
channel.start_consuming()
```

MPI: When lots of flexibility is needed

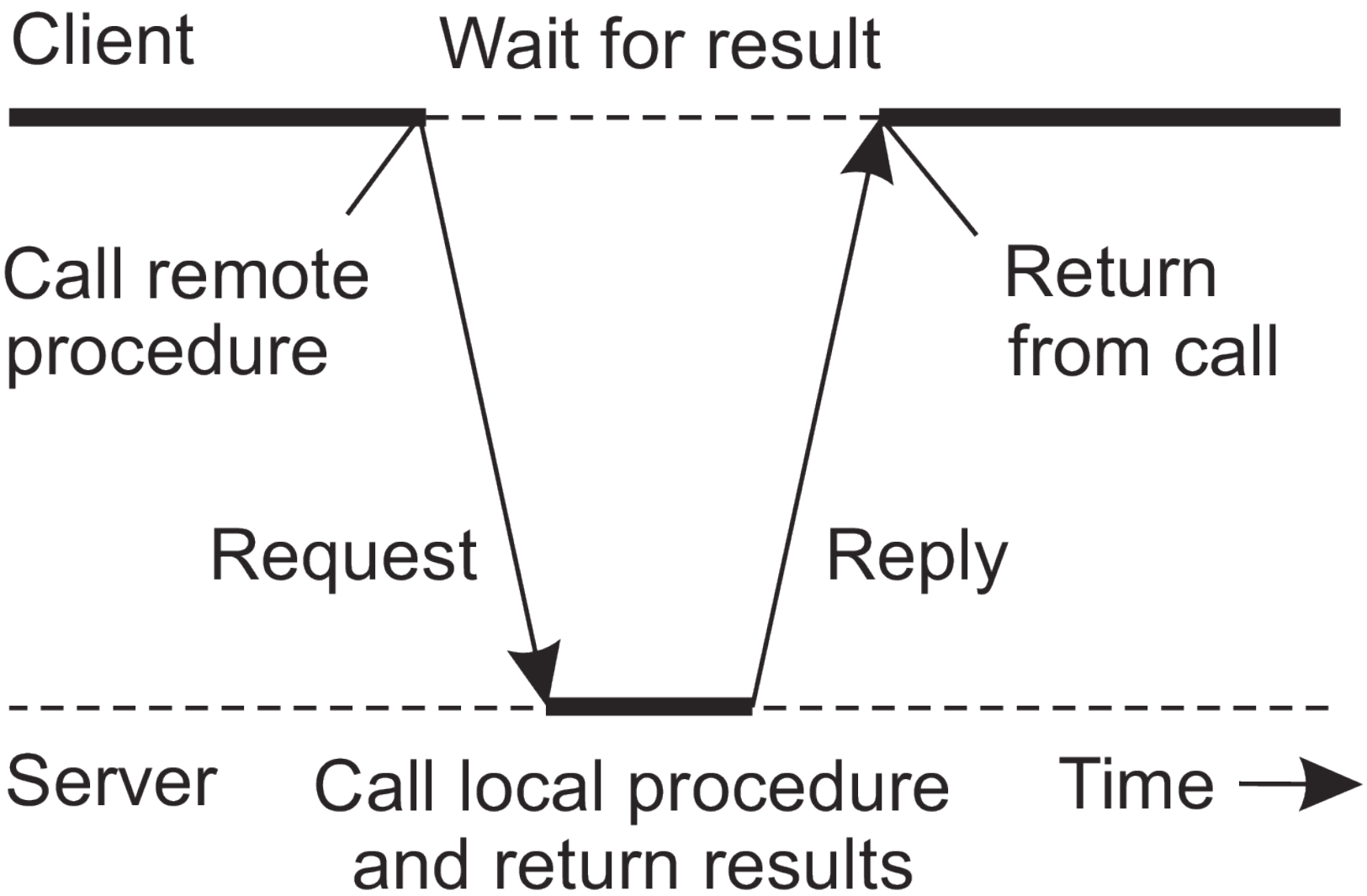
Representative operations

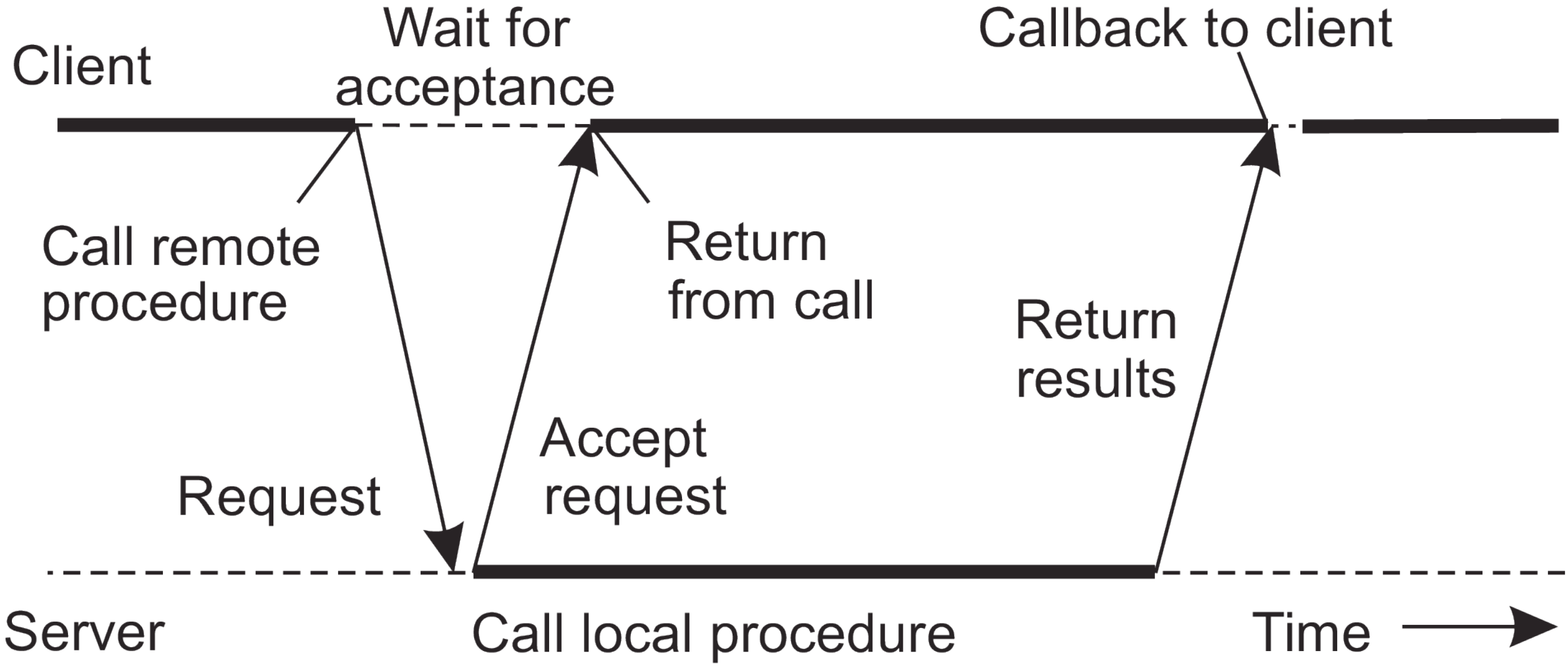
Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, 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

Sockets vs MPI

- Sockets vs MPI
- Conn based on peer integer
- Typed messages, not raw bytes
- Stream vs message oriented
- Not just point to point: collectives, broadcast, etc
- Fast
- Collectives: Scatter, gather, reduce

Extra


Client
Wait for result
Call remote procedure
Return from call
Request
Reply
Server
Call local procedure and return results
Time


Client
Wait for acceptance
Callback to client
Call remote procedure
Return from call
Return results
Request
Accept request
Server
Call local procedure
Time

Pipeline

Worker

```
import zmq, time, pickle, sys

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

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

Message-oriented middleware

Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

Operations

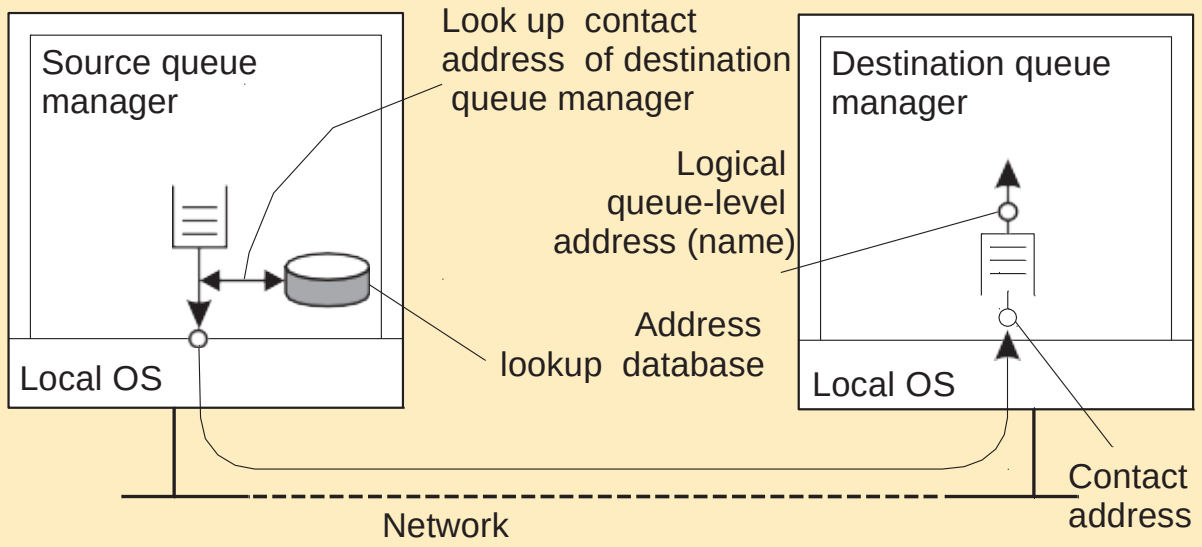
Operation	Description
`put`	Append a message to a specified queue
`get`	Block until the specified queue is nonempty, and remove the first message
`poll`	Check a specified queue for messages, and remove the first. Never block
`notify`	Install a handler to be called when a message is put into the specified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

General model

Queue managers

Queues are managed by queue managers. An application can put messages only into a local queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages.

Routing



Higher-Level Communication

Communication techniques

- TCP/IP protocol stack
- Sockets: OS service for using network communication
  - Low-level
- Basic server model:
  - A process implementing a specific service on behalf of collection of clients.
  - Waits for an incoming request from a client, and ensures that it is taken care of.
  - Waits for another incoming request after responding
- Concurrent servers: Use a dispatcher, which passes the request to a separate process/thread

- RPC’s : Remote Procedure Calls
  - Higher-level than sockets

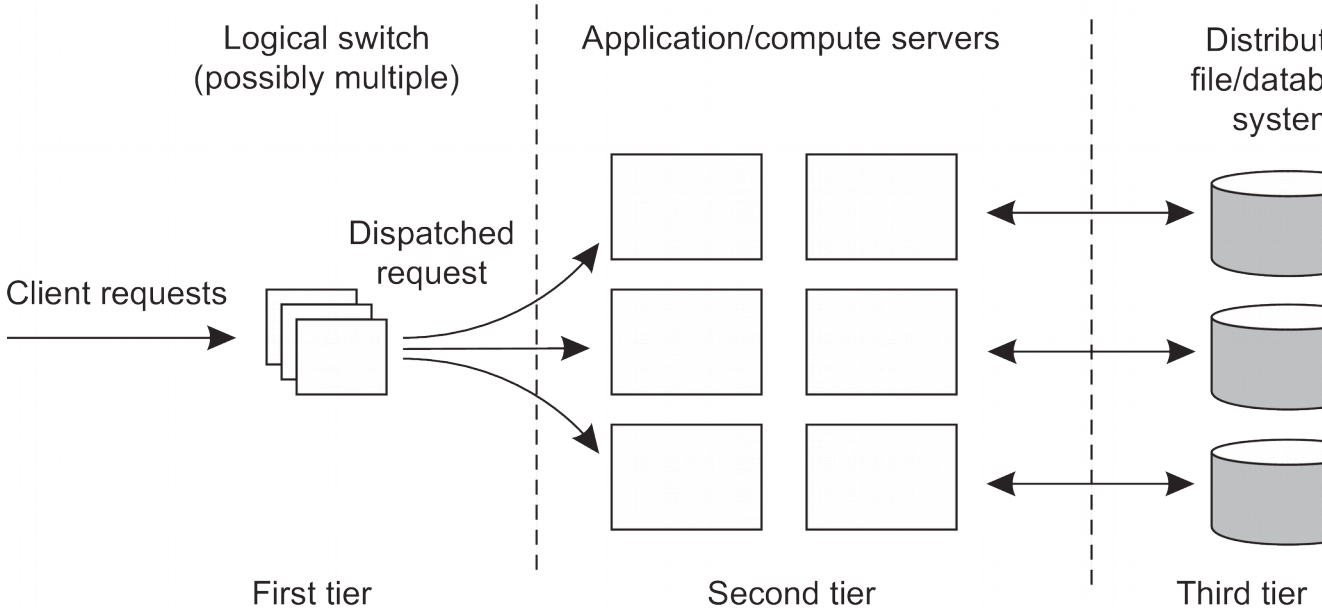
Client/Server

- Stateless servers are a viable design option
- Dont rely on accurate information about the status of a client after having handled a request
  - Don’t record whether a file has been opened (simply close it again aft access)
  - Don’t promise to invalidate a client’s cache
  - Don’t keep track of clients
- Many consequences:
  - Clients and servers are completely independent
  - State inconsistencies due to client or server crashes are reduced
  - Possible loss of performance if server cannot anticipate client behavio (such as prefetching data)

Decoupled Communication

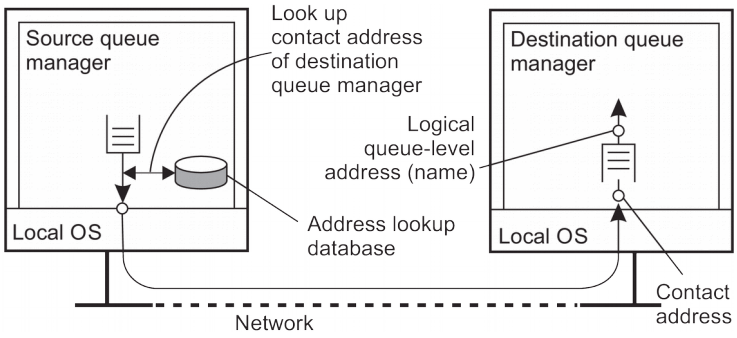
- **Space decoupling**. Interacting parties don’t need to know about each other
- **Time decoupling**. Need not be actively participating/running the same time.
- **Synchronization decoupling**. Sender, receiver should not block
  - Production/consumption do not happen in main flow of control
- Increases scaling, since there is no explicit dependency

Multi-tier Server Architecture



Higher Level Messaging: Message Queues

- Easy to make programming mistakes with sockets
- Popular patterns of usage that can be re-used (like client/server)
- Message Queueing systems provide a higher level of expression



Socket programming is not trivial. Many mistakes. With client/server, we had to open a new socket to respond to client. Lots of times, a higher-level is used. This makes programming the communication easier. One such abstraction is a message queue. Process inserts in a local queue, and its job of queue manager to deliver the message to the queue of the remote process. Note that processes can be communicating with multiple end-points, so the queue manager also needs to do routing of messages.

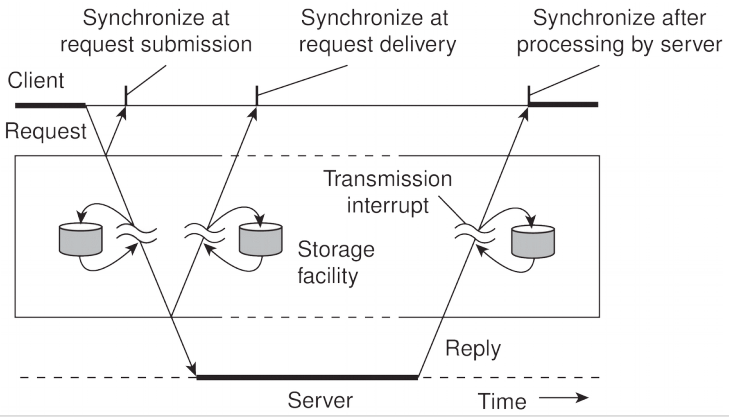
Message Queue Challenges

- How do we handle dynamic components, i.e., pieces that go away temporarily? Do we formally split components into "clients" and "servers" and mandate that servers cannot disappear? What then if we want to connect servers to servers? Do we try to reconnect every few seconds?
- How do we represent a message on the wire? How do we frame data so it's easy to write and read, safe from buffer overflows, efficient for small messages, yet adequate for the very largest videos of dancing cats wearing party hats?
- How do we handle messages that we can't deliver immediately? Particularly, if we're waiting for a component to come back online? Do we discard messages, put them into a database, or into a memory queue?
- How do we handle I/O? Does our application block, or do we handle I/O in the background? This is a key design decision. Blocking I/O creates architectures that do not scale well. But background I/O can be very hard to do right.

IO must be async which is hard because of callbacks. Dynamism. Could do C/S but then different semantics. Low-level aspects of message representation and parsing. Remember that TCP is stream based, and parsing can be a pain. When does a message end?

Persistent Communication

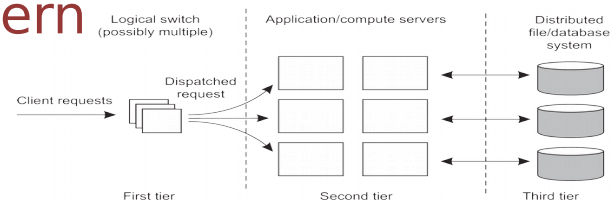
- Socket based: transient communication.
  - Messages sent/rcvd are ephemeral
- Persistent communication: messages are stored by messaging layer

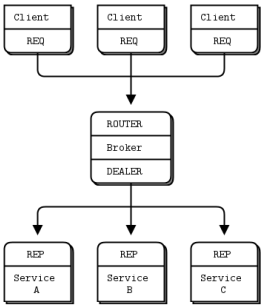


One way to address the challenges is to use persistent communication. Messages

Broker Design Pattern

- Add a message broker between client and server
- Easier to scale client/server
- Clients decoupled from servers
- Only broker is static

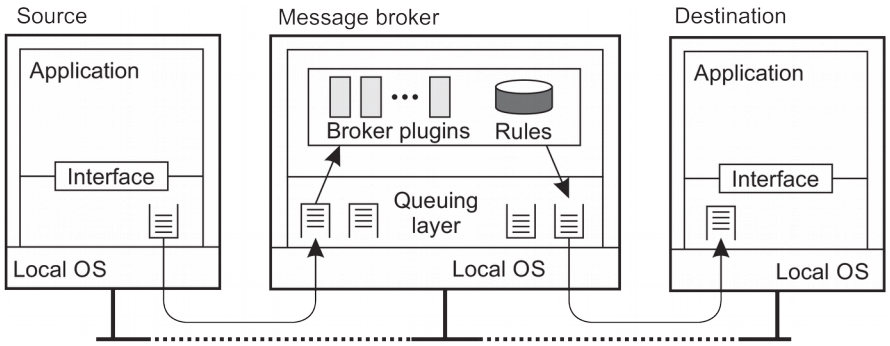




A way to implement persistence is through a message broker

It is similar to load balancers for http-servers. Easier to scale and decouple since only broker is static

Message Broker: General Architecture



Common message brokers: RabbitMQ, Apache Kafka, ...

In general, the brokre works with queue to provide the featres we want. Broker can have rules for routing messages

RabbitMQ and Kafka essentially serve as message brokers

Request-Reply with ZeroMQ

- Higher-level socket-like interface
- No broker required

Server

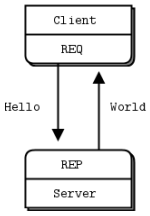
```
1  import zmq
2  context = zmq.Context()
3
4   p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
5   p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
6  s = context.socket(zmq.REP)    # create reply
7                                 socket
8   s.bind(p1)                        # bind socket to
9   s.bind(p2)                        address   # bind
10 while True:                        socket to address
11   message = s.recv()                # wait for incoming
12   if not "STOP" in message:         message   # if not to
13   s.send(message + "*")             stop...
14   else:                             # append "*" to message
15   break                              # else...
                                       # break out of loop and
                                       end
```

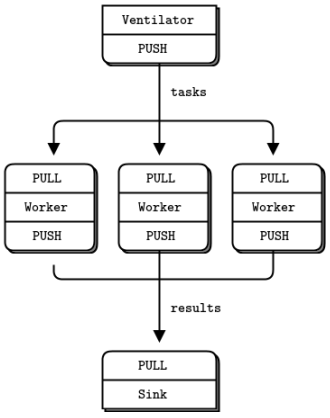
Request-Reply

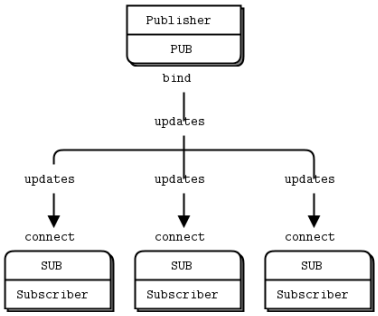
Client

```
1 import zmq
2 context = zmq.Context()
3
4 php = "tcp://"+ HOST +":"+ PORT # how and where to
connect
5 s = context.socket(zmq.REQ) # create socket
6                               # block until
7 s.connect(php)                connected   # send
8 s.send("Hello World")         message
9 message = s.recv()            # block until
10 s.send("STOP")               response   # tell
11 print message                 server to stop  #
                                print result
```

Queue-based communication patterns







ZeroMQ allows multiple communication patterns

Publish-Subscribe

- Need flexible communication models and systems
- Dynamic and decoupled nature of applications
- Point-to-point and synchronous communication is not ideal
  - Leads to rigid and static applications
  - Elastic applications essential in large clouds and data centers (failures, pricing, etc.)
- Publish-Subscribe is one interaction scheme for addressing these problems
- Enables loosely coupled systems
- Fully decouple time, space, and synchronization

Continuing our discussion. Sometimes more flexibility is needed in comm models and dynamic and decoupled. Point to point and sync is not ideal. Apps rigid and static. Cant add nodes. Fault tolerance is harder

Publish-Subscribe basics

- Publishers: Generate events of different types
- Subscribers: Express interest in an event or pattern of events
- Get notified if events match their registered interest
- Producers publish info on a “software bus”
- Usually requires an event-service
  - Provides storage and management for subscription
  - What type of events exist, where to deliver, etc.
  - Also efficient delivery of messages

More Publish-Subscribe

- Subscribers register interest by calling subscribe() on event service, without knowing the sources of the event
- Events pushed via publish()
- Publishers can also advertise the kind of events they will generate in future

Decoupled Communication

- **Space decoupling**. Interacting parties don’t need to know about each other
  - Publishers/subscribers don’t need to hold references to each other
- **Time decoupling**. Need not be actively participating/running the same time.
  - Subscribers can get disconnected and be notified later
- **Synchronization decoupling**. Sender, receiver should not block
  - Publishers not blocked while producing events
  - Subscribers can get async notified via callbacks
  - Production/consumption do not happen in main flow of control
- Increases scaling, since there is no explicit dependency

How coupled are other communication patterns?

- Message Passing: Producer and consumer are coupled in time and space
  - Transient communication, not persistent
- RPC
  - Space coupling: invoking object holds remote reference to each invok
- Async RPC: fire and forget, Futures and Promises based programming
  - Decoupled

Publish-subscribe in ZeroMQ

Server

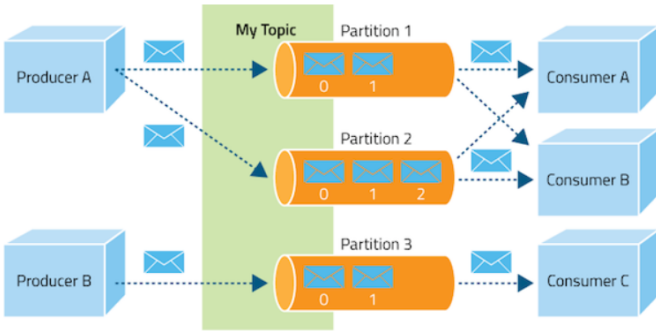
```
1 import zmq, time
2
3 context = zmq.Context()              # create a publisher socket
4 s = context.socket(zmq.PUB)          # how and where to
5 p = "tcp://"+ HOST +":"+ PORT        communicate   # bind socket
6 s.bind(p)                            to the address
7 while True:
8    time.sleep(5)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

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
6 s.connect(p)                         communicate   # connect to
7 s.setsockopt(zmq.SUBSCRIBE,          the server
"TIME")                                # subscribe to TIME
8 for i in range(5):  # Five iterations messages
10   time = s.recv()  # receive a message
11   print time
```

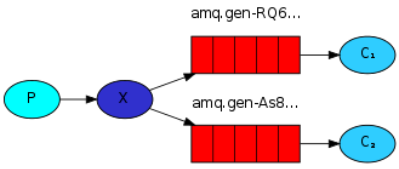
RabbitMQ, Kafka

- Modern pub/sub systems with persistent broker/agent
- Some “broker” should always be running
- Message routing based on key/topic
- Message Queues can also be persistent/durable



RabbitMQ Publish

```
import pika
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='fanout')
message = 'info:Hello'
channel.basic_publish(exchange='logs', routing_key='', body=message)
connection.close()
```



RabbitMQ Subscribe

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='fanout')   #broadcasts to all
result = channel.queue_declare(queue='', exclusive=True)  #close if no consumers left
queue_name = result.method.queue
channel.queue_bind(exchange='logs', queue=queue_name)
print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] %r" % body)

channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
channel.start_consuming()
```

MPI: When lots of flexibility is needed

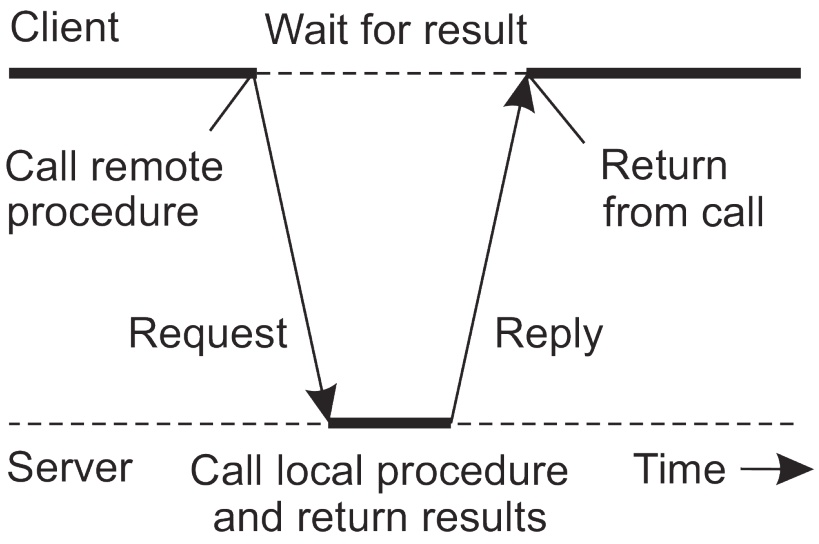
Representative operations

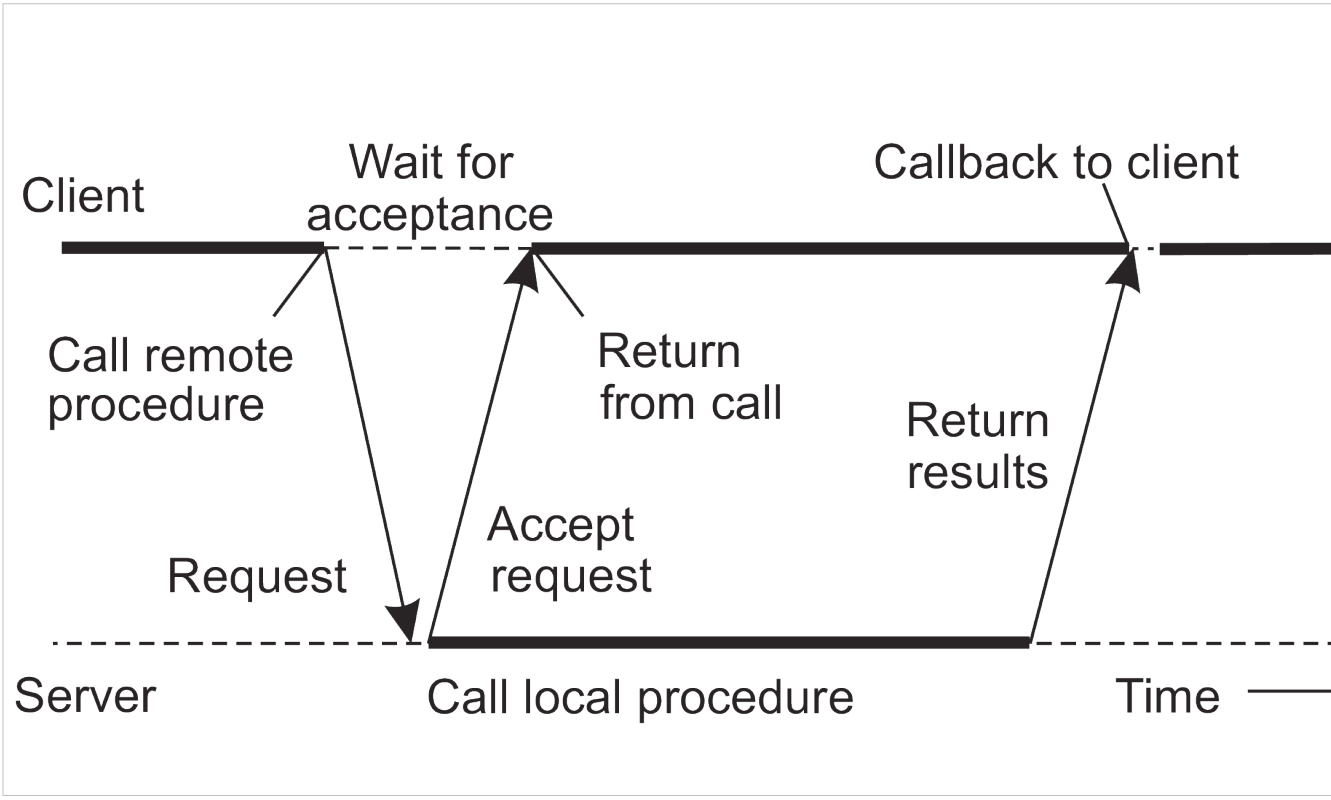
Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, 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

Sockets vs MPI

- Sockets vs MPI
- Conn based on peer integer
- Typed messages, not raw bytes
- Stream vs message oriented
- Not just point to point: collectives, broadcast, etc
- Fast
- Collectives: Scatter, gather, reduce

Extra


Client
Wait for result
Call remote procedure
Return from call
Request
Reply
Server
Call local procedure and return results
Time


Client
Wait for acceptance
Callback to client
Call remote procedure
Return from call
Return results
Accept request
Request
Server
Call local procedure
Time

Pipeline

Worker

```
1 import zmq, time, pickle, sys
2
3 context = zmq.Context()
4 me = str(sys.argv[1])
5 r = context.socket(zmq.PULL)     # create a pull socket
6 p1 = "tcp://"+ SRC1 +":"+       # address first task
PORT1   7 p2 = "tcp://"+ SRC2      source  # address second
+":"+ PORT2                        task source   # connect
8 r.connect(p1)                    to task source 1  #
9 r.connect(p2)                    connect to task source 2
10
11 while True:                     # receive work from a
12   work =                        source   # pretend to
pickle.loads(r.recv())             work
13   time.sleep(work[1]*0.01)
```

Message-oriented middleware

Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

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

General model

Queue managers

Queues are managed by queue managers. An application can put messages only into a local queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages.

Routing