COMP S362F Unit 11 Message queues

Kendrew Lau

Contents

- Messaging and message queues
- Programming IPC pipes
- Programming ZeroMQ

Messaging and message queues

Inter-process communication (IPC)

- There are different ways for concurrent units of an application to coordinate and communicate
- Sharing variables – only for threads
  - Not really inter-process
- Sharing memory – mainly for processes
  - Supported by operating systems
- Passing messages – mainly for processes
  - Supported by operating systems, software libraries, or messaging servers (middleware)

Messaging

- *Messaging* refers to transferring messages to communicate between 2 or more processes (or distributed systems)
    - One process sends a message, and another process receives the message
    - Communication is mostly asynchronized, but may also be synchronized
- In *asynchronized mode*, the sender doesn't wait for the receiver to accept the message but proceeds to other tasks
    - Like text messages, WhatsApp
- In *synchronized mode*, the sender waits for the receiver to accept the message
    - Like telephone calls

Message-oriented middleware (MOM)

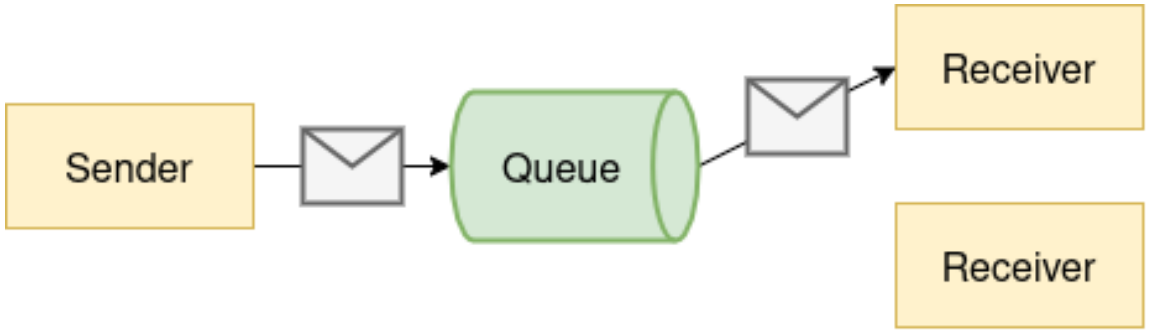
- MOM is application servers that support sending and receiving messages
- Major features
  - Asynchronicity: persisting a message until a client receives it later
  - Routing: customized forwarding of messages according to types and contents
  - Transformation: modifying message formats and contents

Message queues

- A *message queue (MQ)* is a first-in-first-out (FIFO) buffer of messages for communicating data between processes
- Many people use “message queues” to refer to messaging in general
- Two common and classic messaging models/patterns
    - Point-to-point messaging
    - Publish/subscribe messaging

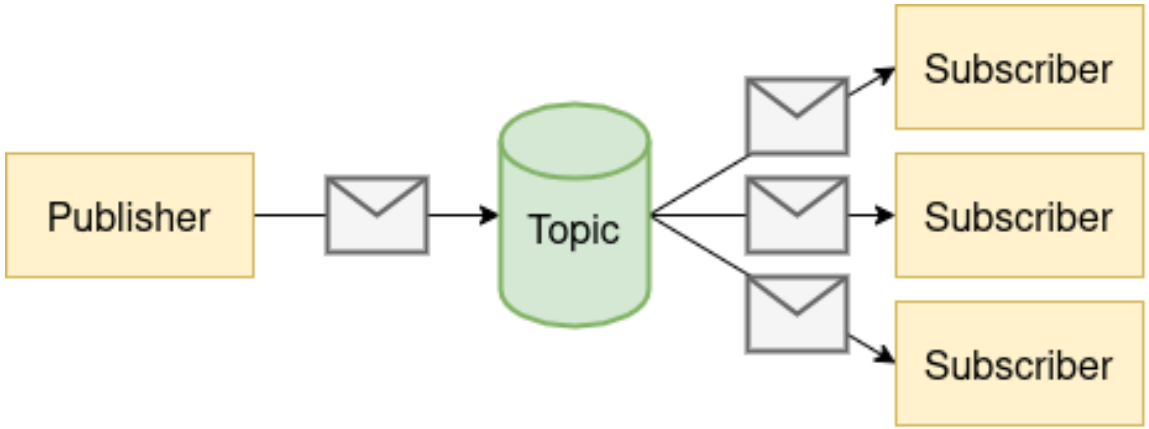
Point-to-point (p2p) messaging

- The sender places a message on a queue with optional expiry
- One receiver obtains the message from the queue
  - And the message is removed from the queue
- E.g. online e-commerce order processing
  - A customer order is placed on a message queue; a worker fetches the order from the queue and handles it



Publish/subscribe (pub/sub) messaging

- Multiple users subscribe to a topic
- When a publisher posts a message, it is visible to all subscribers
    - The system may be set up to (re)send a message to a previously-offline subscriber
- E.g. WhatsApp chat group, mail newsletter



Programming IPC pipes

IPC pipes

- A *pipe* is like a message queue between 2 processes
  - For inter-process communication
  - Implemented in Unix-like and Windows operating systems
- Python supplies two APIs for pipes
  - `os.pipe`: lower level, unidirectional
  - `multiprocessing.Pipe`: higher level, bidirectional

Using multiprocessing.Pipe

- Creating a pipe
  - `Pipe(duplex=True)`: return a pair of `Connection` objects representing the ends of a pipe
- Methods of `Connection`
  - `send(obj)`: send an object to the other end of the connection
  - `recv()`: return an object sent from the other end of the connection, potentially blocking
  - `close()`: close the connection

Simple pipe example

```
# File: pipe_hello.py
import multiprocessing

def p1(conn):
    pname = multiprocessing.current_process().name
    conn.send("hello")
    print(pname, "sent: hello", flush=True)
    obj = conn.recv()
    print(pname, "received:", obj, flush=True)

def p2(conn):
    pname = multiprocessing.current_process().name
    obj = conn.recv()
    print(pname, "received:", obj, flush=True)
    conn.send("world")
    print(pname, "sent: world", flush=True)

if __name__ == "__main__":
    conn1, conn2 = multiprocessing.Pipe()
    multiprocessing.Process(target=p1, args=(conn1,)).start()
    multiprocessing.Process(target=p2, args=(conn2,)).start()
```

```
Process-1 sent: hello
Process-2 received: hello
Process-2 sent: world
Process-1 received: world
```

Sending a sequence of values

- To send a sequence of values, how to indicate the end of sequence?
- Approach 1: sending and detecting a *sentinel*, i.e. “stopping value”
  - A sentinel is a special value (invalid data value) that indicates the end of sequence
- Approach 2: closing connection and detecting `EOFError`
  - Method `recv()` raises `EOFError` when no data is available and the other end is closed
  - Need to close that connection in both processes!

Sentinel example

```python
# File: pipe_sentinel.py
from multiprocessing import Pipe, Process
import random

SENTINEL = -1

def child(conn):
    while True:
        obj = conn.recv()
        if obj == SENTINEL:
            break
        print(obj, end=" ", flush=True)

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
    Process(target=child, args=(child_conn,)).start()
    for i in range(10):
        parent_conn.send(random.randint(1, 100))
    parent_conn.send(SENTINEL)
```

```
9 53 91 67 48 56 23 48 59 56
```

Using unidirectional pipes

- The two connections of a unidirectional pipe are for input and output

```
conn_in, conn_out = multiprocessing.Pipe(False)
...
conn_out.send(42)
obj = conn_in.recv()
```

- Data can flow through multiple processes linked by pipes
    - Each process reads data from the input connection of a pipe, and writes data to the output connection of another a pipe

Unidirectional pipe example

```
# File: pipe_unidir.py
from multiprocessing import Pipe, Process

SENTINEL = -1

def divisible(conn_in, conn_out, num):
    while True:
        obj = conn_in.recv()
        if obj == SENTINEL:
            conn_out.send(SENTINEL)
            break
        if obj % num == 0:
            conn_out.send(obj)
```

Unidirectional pipe example (cont.)

```
        main --> conn1_out - (pipe#1) - conn1_in --> divisible(2)
divisible(2) --> conn2_out - (pipe#2) - conn2_in --> divisible(5)
divisible(5) --> conn3_out - (pipe#3) - conn3_in --> main
```

```
# File: pipe_unidir.py (cont.)
if __name__ == "__main__":
    conn1_in, conn1_out = Pipe(False)
    conn2_in, conn2_out = Pipe(False)
    conn3_in, conn3_out = Pipe(False)
    Process(target=divisible, args=(conn1_in, conn2_out, 2)).start()
    Process(target=divisible, args=(conn2_in, conn3_out, 5)).start()
    for i in range(100):
        conn1_out.send(i)
    conn1_out.send(SENTINEL)
    while True:
        obj = conn3_in.recv()
        if obj == SENTINEL:
            break
        print(obj, end=" ", flush=True)
```

```
0 10 20 30 40 50 60 70 80 90
```

Programming ZeroMQ

ZeroMQ

- *ZeroMQ* is a universal messaging library
  - Also known as *ØMQ*, *0MQ*, or *zmq*
  - Code in any language on any platform
  - https://zeromq.org/
- Zero means minimalism
  - Zero broker, zero latency, zero administration, zero cost, zero waste, ...
  - You can write a broker specifically for your application
- A familiar socket-based API, with messaging functionality
- To install the Python library: `pip install pyzmq`

Built-in core ZeroMQ patterns

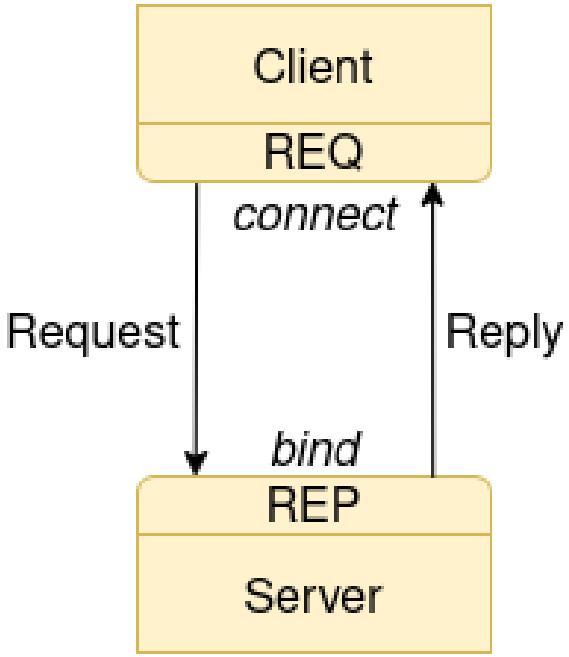
- Request-reply
  - Connect a set of clients to a set of services
  - A remote procedure call and task distribution pattern
- Pub-sub
  - Connect a set of publishers to a set of subscribers
  - A data distribution pattern
- Pipeline
  - Connect nodes in a fan-out/fan-in pattern that can have multiple steps and loops
  - A parallel task distribution and collection pattern
- Exclusive pair
  - Connect two sockets exclusively
  - A pattern for connecting two threads in a process, not to be confused with “normal” pairs of sockets

Major PyZMQ APIs

- `ctx = zmq.Context()`: create a 0MQ context
- `s = ctx.socket(socket_type)`: create a 0MQ socket
- Methods of a 0MQ socket
  - `s.bind(addr)`: bind the socket to an address
  - `s.connect(addr)`: connect to a remote 0MQ socket
  - `s.close()`: close a 0MQ socket
  - `s.send(bytes)` `s.recv()`: send and receive binary data
  - `s.send_string(string)` `s.recv_string()`: send and receive a Unicode string
  - `s.send_json(obj)` `s.recv_json()`: send and receive an object using JSON format

Request-reply pattern

- Connect a set of clients to a set of services
- A remote procedure call and task distribution pattern



Request-reply hello server

```
# File: hello_server.py
import time
import zmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    message = socket.recv_string()
    time.sleep(0.1)
    socket.send_string(message + " world")
```

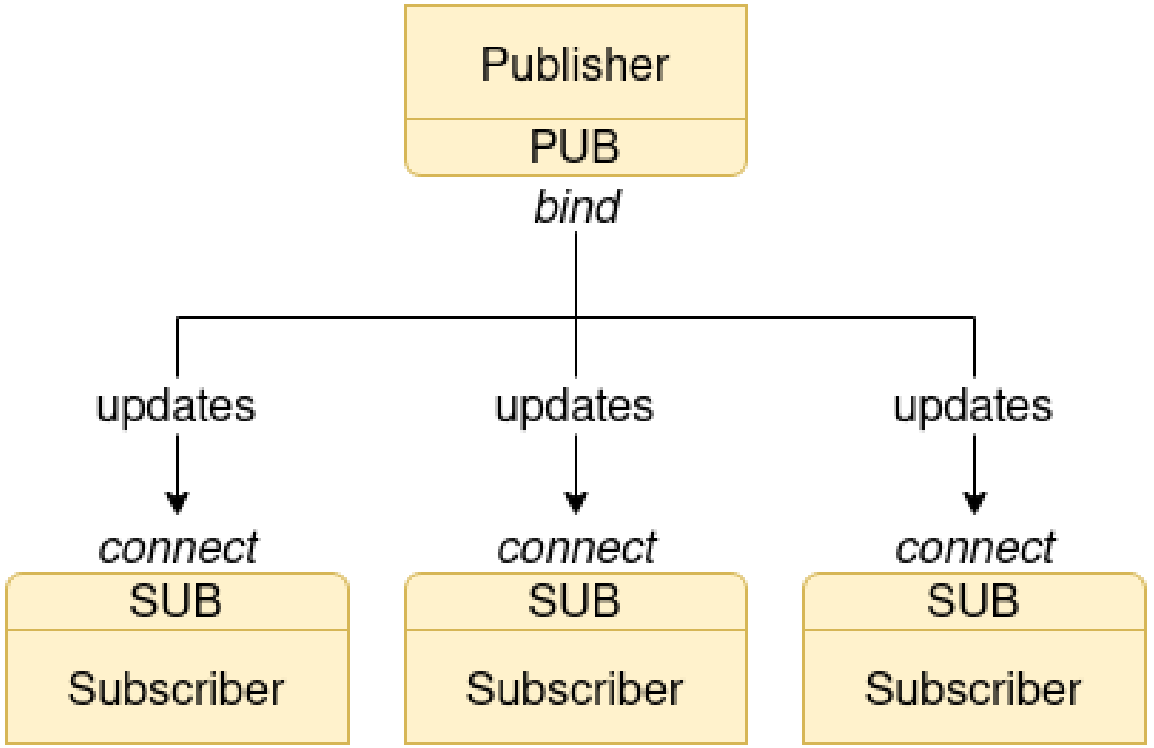
Request-reply hello client

```python
# File: hello_client.py
import zmq

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

for i in range(10):
    request = f"#{i} hello"
    socket.send_string(request)
    print("Client sent:", request)
    reply = socket.recv_string()
    print("Client received:", reply)
```

```
Client sent: #0 hello
Client received: #0 hello world
Client sent: #1 hello
Client received: #1 hello world
...
Client sent: #9 hello
Client received: #9 hello world
```

Pub-sub pattern

- Connect a set of publishers to a set of subscribers
- A data distribution pattern



Pub-sub weather server

```
# File: weather_server.py
import zmq
import datetime, random, time

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5556")

while True:
    now = str(datetime.datetime.now())
    temperature = random.randint(10, 35)
    message = { "time": now, "temperature": temperature }
    socket.send_json(message)
    time.sleep(1)
```

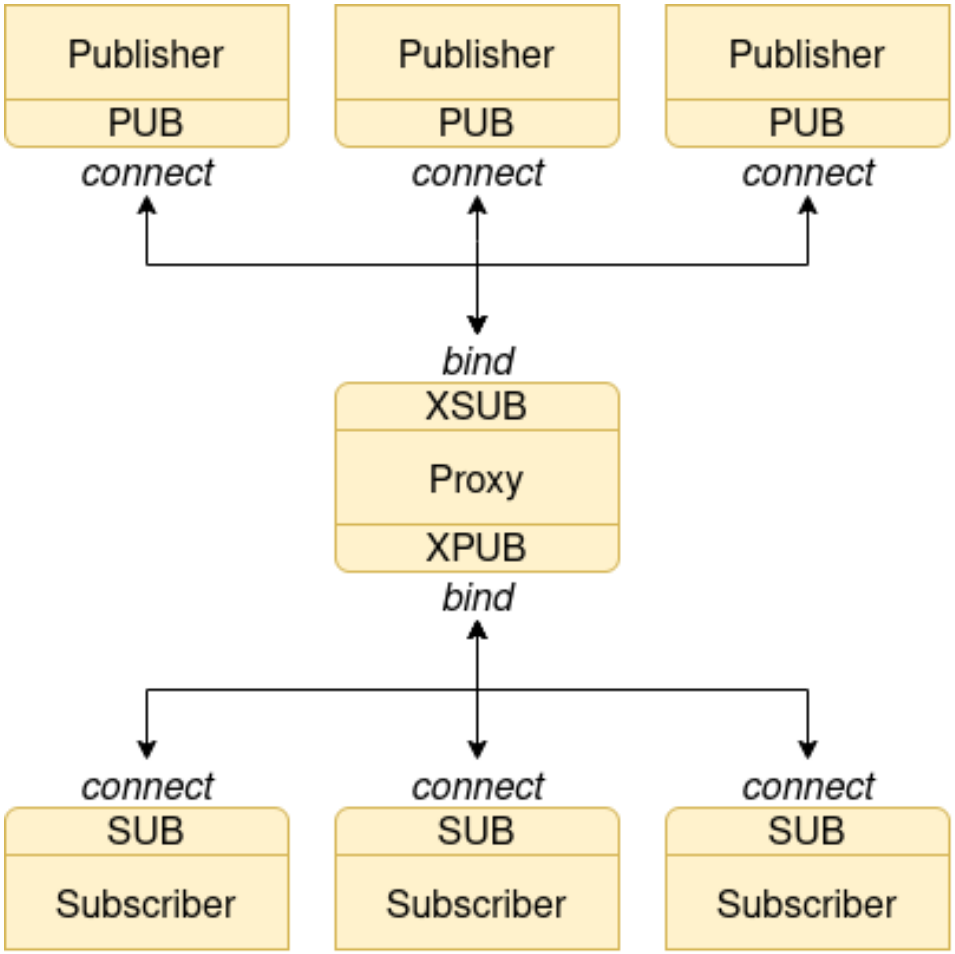
Pub-sub weather client

```python
# File: weather_client.py
import zmq

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect("tcp://localhost:5556")
# set message prefix filter; "" accepts all messages
socket.setsockopt_string(zmq.SUBSCRIBE, "")

message = socket.recv_json()
print(message)
```

```
{'time': '2021-12-02 22:12:21.340325', 'temperature': 13}
```

Extended pub-sub pattern

- Connect between a set of publishers and a set of subscribers
- An intermediary, message broker, stateless message switch



Extended pub-sub chat server

```python
# File: chat_server.py
import threading
import zmq

context = zmq.Context()
xpub = context.socket(zmq.XPUB)
xpub.bind("tcp://*:5555")
xsub = context.socket(zmq.XSUB)
xsub.bind("tcp://*:5556")

def forward(socket1, socket2):
    while True:
        message = socket1.recv()
        socket2.send(message)

# Demo only; 0MQ sockets not thread-safe
threading.Thread(target=forward, args=(xpub, xsub)).start()
threading.Thread(target=forward, args=(xsub, xpub)).start()
```

Extended pub-sub chat client

```
# File: chat_client.py
import threading
import zmq

def receive(context):
    sub = context.socket(zmq.SUB)
    sub.connect("tcp://localhost:5555")
    sub.setsockopt_string(zmq.SUBSCRIBE, "")
    while True:
        message = sub.recv_string()
        print(message.rjust(70))

context = zmq.Context()
threading.Thread(target=receive, args=(context,)).start()

pub = context.socket(zmq.PUB)
pub.connect("tcp://localhost:5556")
while True:
    message = input()
    pub.send_string(message)
```

```
hello
                                                                  hello
                                                             hi there
```

The end

Appendix: Closing connection example

```python
# File: pipe_eoferror.py
from multiprocessing import Pipe, Process
import random

def child(child_conn, parent_conn):
    parent_conn.close()
    while True:
        try:
            obj = child_conn.recv()
        except EOFError:
            break
        print(obj, end=" ", flush=True)

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
    Process(target=child, args=(child_conn, parent_conn)).start()
    for i in range(10):
        parent_conn.send(random.randint(1, 100))
    parent_conn.close()
```

```
70 72 32 91 14 58 75 94 97 42
```