Producer Consumer in Java

```
class prodcon{
   int N = 80;
   int buffer[] = new int[N];
   int numItems = 0;
   int in = 0, out = 0;

   synchronized void produce(int item)
   {
      while(numItems == N)
      {
         try {
            wait();
         }catch(Exception exc){}
      }
      buffer[in] = item;
      in = (in+1)%N;
      numItems++;
      notify();
   }

   synchronized int  consume()  {
      while(numItems == 0)   {
         try {
            wait();
         } catch(Exception exc){}
      }
      int retItem = buffer[out];
      out = (out+1)%N;
      numItems--;
      notify();
      return out;
   }
}

public static void main(String args[])  {
      prodcon pc = new prodcon();
      Thread producer = new Thread() {
         public void run() {
            for(int i = 0; i < 100; i++)   {
               pc.produce(i);
            }
         }
      };
      Thread consumer = new Thread() {
         public void run() {
            for(int i = 0; i < 100; i++)  {
               int retItem = pc.consume();
            }
         }
      };
      producer.start();
      consumer.start();
      try {
         producer.join();
         consumer.join();
      }catch(Exception exc){}
   }
}
```

Producer Consumer in Java - alternate

```
import java.util.concurrent.locks.*;
class prodcon1 {
    int N = 80;
    int buffer[] = new int[N];
    int numItems = 0;
    int in = 0, out = 0;
    ReentrantLock lock = new ReentrantLock();
    Condition full = lock.newCondition();
    Condition empty = lock.newCondition();
    void produce(int item)  {
        lock.lock();
        while(numItems == N) {
            try {
                empty.await();
            }catch(Exception exc){}
        }
        buffer[in] = item;
        in = (in+1)%N;
        numItems++;
        full.signal();
        lock.unlock();
    }
    int  consume()  {
        lock.lock();
        while(numItems == 0)  {
            try {
                full.await();
            } catch(Exception exc){}
        }
        int retItem = buffer[out];
        out = (out+1)%N;
        numItems--;
        empty.signal();
        lock.unlock();
        return out;
    }
}
```

Producer Consumer in Python

```
import threading

class ProdCon():
  def __init__(self):
    self.buffer = []
    self.numItems = 0
    self.lock = threading.RLock()
    self.full = threading.Condition(lock =
self.lock)
    self.empty = threading.Condition(lock =
self.lock)
    self.N = 20

  def produce(self, item):
    self.lock.acquire()
    while len(self.buffer) == self.N:
        self.empty.wait()
    self.buffer.append(item)
    self.full.notify()
    self.lock.release()
```

```
 def consume(self):
   self.lock.acquire()
   while len(self.buffer) == 0:
       self.full.wait()
   retItem = self.buffer.pop()
   self.empty.notify()
   self.lock.release()
   return retItem

class Producer(threading.Thread):
  def __init__(self, prodcon):
   threading.Thread.__init__(self)
   self.prodcon = prodcon

  def run(self):
    for i in range(100):
      self.prodcon.produce(i)
```

```
class Consumer(threading.Thread):
  def __init__(self, prodcon):
    threading.Thread.__init__(self)
    self.prodcon = prodcon

  def run(self):
    for i in range(100):
      retItem = self.prodcon.consume()


prodcon = ProdCon()
producer = Producer(prodcon)
consumer = Consumer(prodcon)
consumer.start()
producer.start()
producer.join()
consumer.join()
```

Message Passing Synchronization

- It represent an alternative way to shared memory solutions.
  - A mandatory solution when actors reside on different machines not sharing memory
- In its simplest, and most abstract, form a message-passing mechanism involves two basic primitives:
  - a **send** primitive, which sends a certain amount of information, called a message, to another process;
  - a **receive** primitive, which allows a process to block waiting for a message to be sent by another process, and then retrieve its contents.
- Even if this definition still lacks many important details it is already clear that the most apparent effect of message passing primitives is to transfer a certain amount of information from the sending process to the receiving one.
- At the same time, the arrival of a message to a process also represents a synchronization signal because it allows the process to proceed after a blocking receive.
- The last important requirement of a satisfactory interprocess communication mechanism, mutual exclusion, is not a concern here because messages are never shared among processes, and their ownership is passed from the sender to the receiver when the message is transferred. In other words, the mechanism works as if the message were instantaneously copied from the sender to the receiver
  - even if real-world message passing systems do their best to avoid actually copying a message for performance reasons

Main design choices in message passing

1. For a sender, how to identify the intended recipient of a message. Symmetrically, for a receiver, how to specify from which other processes it is interested in receiving messages. In more abstract terms, how a process naming scheme must be defined.
   a. how the send and receive primitives are associated to each other;
   b. their symmetry (or asymmetry).

2. The synchronization model, that is, under what circumstances communicating processes shall be blocked, and for how long, when they are engaged in message passing.

3. How making a message content consistent (endianity and pointers)

Direct vs indirect naming

- The most straightforward approach is for the sending process to name the receiver directly
  - For example, in a IP communication the sender shall specify the Internet Protocol (IP) address and port of the intended receiver.

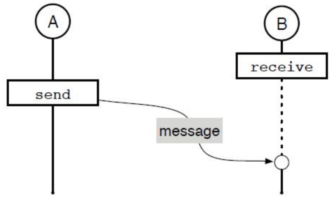
- Considering the receiver, direct naming is not so straightforward
  - Normally the receiver provided a service that may be accessed by different clients
  - After a connection has been established sender and receiver exchange messages and can therefore swap their roles
  - Both must adhere to a **communication protocol** in order to achieve a consistent communication

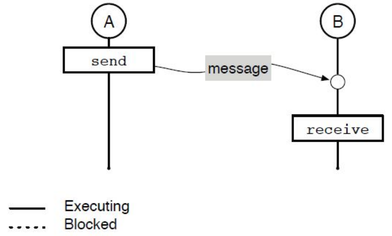
- Indirect naming means that the sender does not address the receiver explicitly.
  - For example in a multicast communication, a set of receivers may receive the messages sent by the server, that may also be unaware of the identity of the recipients

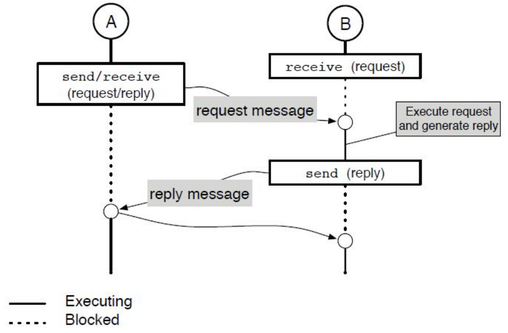
- **Publish-Subscribe** is a common pattern in this context
  - A Publisher offers some kind of service
  - A Subscriber can subscribe for a given service and shall receive messages issued by the publisher
  - More than one subscriber may be registered and thew publisher may be unaware of this, especially when association between publishers and subscribers is implemented by a separate **broker**.

Message Synchronization

- The receiver normally needs to synchronize to the message receipt
  - Asynchronous receive shall define a callback to be called asynchronously when a message has been received
- Normally send() call returns soon, unless internal message queuing reaches a given limit
- In randezvous schema, the sender synchronizes with the message reception reception
- Normally randezvous schema are implemented with a acknowledge message sent back by the receiver
- The communication protocol specifies the messages to be sent and received in order to achieve the required synchronization







Message Buffering

- Very often the send() routine will return before the message has been received
- If communication occurs within the same computer, this means temporarily storing exchanged packets in memory (managed by the OS)
- If communication involves different computers, packets may be stored in any point of the path between the sender and the receiver
- Considering network communication, messages are often buffered in routers before being dispatched at the right port, based on the message header and the routing information (store and forward)
- Other routers start dispatching the message as soon as the header has been received and the destination can be computed (cut-through)
- Message buffering is normally transparent to the sender and the receiver and therefore no assumption can be done about message transfer time.
- If messages are used for synchronization, the use of Acknlowledge return messages is required

Pros and Cons of Message Buffering

- Having a large buffer between the sender and the receiver decouples the two processes and, on average, makes them less sensitive to any variation in execution and message passing speed.

- Therefore it increases the likelihood of executing them concurrently without unnecessarily waiting for one another.

- The interposition of a buffer increases the message transfer delay and makes it less predictable.

- For some synchronization models, the amount of buffer space required at any given time to fulfill the model may depend on the processes' behavior and be very difficult to predict.

- For the purely asynchronous model, the maximum amount of buffer space to be provided by the system may even be unbounded in some extreme cases. This happens, for instance, when the sender is faster than the receiver so that it systematically produces more messages than the receiver is able to consume

Reliable vs Unreliable message transfer

- The transmission of a message is subject to several possible communication errors
- Effect of communication errors can be:
  - Message corruption
  - Messages lost
  - messages duplicated

- Reasons for for communication errors can be:
  - transmission errors
  - message discarded in internal router buffers when the traffic is too high

- Reliability can be granted on not depending on the communication protocol adopted
  - TCP/IP guarantees reliable delivery and non corrupted messages
  - UDP guarantees only that received messages are not corrupted (checksum handled in the protocol) but no guarantee is given about message delivery, i.e. messages can be lost. For Local Area Networks, the reason for the loss of a message is exclusively internal router buffer overrun

Reliability vs real-time communication

- Real-time communication requires that a message shall be successfully received within a given amount of time.
- Reliability in communication requires detecting whether messages are lost...
    - Normally assigning an increasing timestamp value to each message so that missed messages can be detected checking timestamp
- ...and handling the message loss
    - Normally by requesting the missed message
- However this added reliability layer introduce a unbounded delay even if the transmission time is bounded, therefore breaking real-time requirements
- On the other side, loss of messages cannot often be avoided for sure
- This has to be taken into account in communication for real-time distributed systems
- Sometimes a (limited) loss of messages can be tolerated, e.g. when then system is performing closed loop control
    - Missing messages can be considered an added noise in sensor readout or actuator command and may be tolerated provided the overall controllow introduces a stability margin large enough
- Moreover, for LAN communication, by insulating the system from external (unpredictable) traffic it is in general possible to make sure that not internal buffer overrun occurs in the involved routers.
- For this reason, UDP is often adopted as communication protocol in real-time systems.

Message content consistency

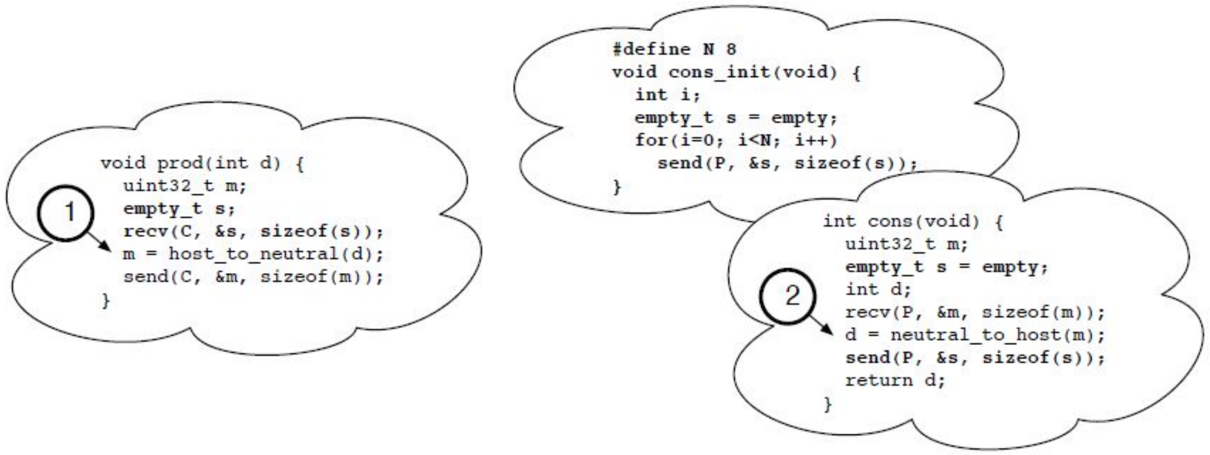
- Not always the bytes forming a message bring consistent information, especially when the receiver resides on a different machine

- A common error is due to the endianity that may be different between the machines involved in communication.
  - char arrays always works, but multybyte numbers can be interpreted differently (big/little endian issue)
  - E.g. the IP socket layer defines a default endianity (big endian) and it is up to the sender/received properly swap bytes

- Another issue is related to the usage of pointers in data structures because
  - This affects also communication on the same machine because different processes have a different address space (virtual addresses)
  - Pointers can be safely exchanged when communication among threads of the same process occurs

- A common practice when sending complex data structures is to serialize them in a sequence of bytes
  - If the structure contains internal pointers (e.g. a linked queue) the pointers are normally replaced by an offset from the first byte of the buffer in serialization and replaced by pointers in deserialization

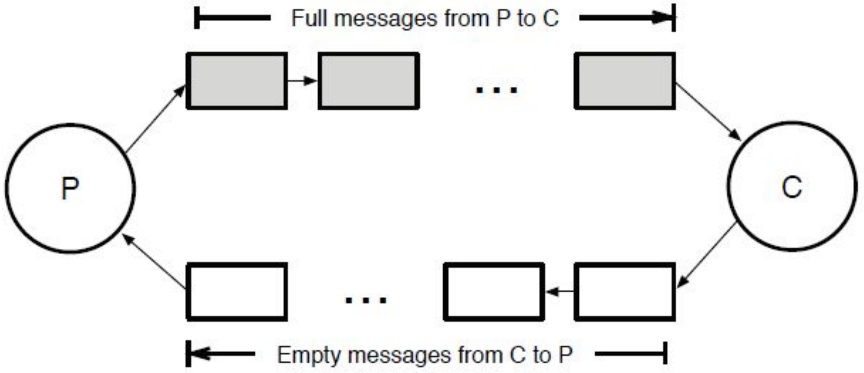
Producer Consumer implemented with messages



```
void prod(int d) {
  uint32_t m;
  empty_t s;
  recv(C, &s, sizeof(s));
  m = host_to_neutral(d);
  send(C, &m, sizeof(m));
}
```

```
#define N 8
void cons_init(void) {
  int i;
  empty_t s = empty;
  for(i=0; i<N; i++)
    send(P, &s, sizeof(s));
}
```

```
int cons(void) {
  uint32_t m;
  empty_t s = empty;
  int d;
  recv(P, &m, sizeof(m));
  d = neutral_to_host(m);
  send(P, &s, sizeof(s));
  return d;
}
```



The programming model of TCP/IP

- In the programming model of TCP/IP, a socket must be established between the client and and the server.

- Once the socket has been established, client and servers can communicate, i.e. send and receive messages, that are ensured to be correctly delivered
  - Both client and server must adhere to a given protocol for meaningful communication
    - e.g if both issue e receive() call, they stop forever

- In order to establish a connection (socket), the client must specify the Internet Protocol (IP) address and the port number of the target server.

- The server instead is not aware of potential client until a connection has been established. The server therefore issues a accept() command that shall return when a client requested the establishment of a communication. At this point both client and server will receive a socket to be used for further read() and write() operations

- In order to be able to serve more than one client concurrently, the server shall start a new thread/process every time the accept() call returns, passing the returned socket to the task

The programming model of UDP

- Unlike TCP/IP, the programming model is message oriented, i.e. a UDP datagram (normally limited to 64Kbytes) can be sent and received

- Still in this case a socket must be created and used for sending and receiving datagrams, however in this case the successful socket creation does not imply the establishment of a connection (in this case the socket is in ‘unconnected’ state)

- Messages are sent via sendto() routine whose arguments shall specify the IP address and port number of the received.

- The a task wants to receive a datagram, it will call recv() or recvfrom(), after binding the socket to a port number
  - recv() shall return when a message has been received by any sender addressing this IP and port.
  - recvfrom() shall return when a message from the specified sender has been received

- UDP allows also the implementation of publish-subscribe pattern using UDP multicast.
  - In this case it is possible to register to a UDP multicast address, so that every message sent to this multicast address shall be received by all registered listeners