1.-

```
public class BufferBroadcastUn {

        protected Object espai;
        protected boolean[] disponible;
        protected Lock lock;
        protected Condition productor, consumidor;
        protected int num_consumidors, consumits;

        public BufferBroadcastUn(int num_consumidors){
                disponible = new boolean[num_consumidors];
                lock = new new ReentrantLock();
                productor = lock.newCondition();
                consumidor = lock.newCondition();
                this.num_consumidors = num_consumidors;
                consumits = num_consumidors;
        }

        public void putValue(Object val){
                lock.lock();
                if(consumits < num_consumidors){
                        productor.await();
                }
                espai = val;
                consumits = 0;
                for(int i=0; i<disponible.length; i++){
                        disponible[i] = true;
                }
                consumidor.signalAll();
                lock.unlock();
        }

        public Object getValue(int id){
                Object tmp;
                lock.lock();
                if(!disponible[id]){
                        consumidor.await();
                }
                tmp = espai;
                disponible[id] = false;
                consumits++;
                if(consumits==num_consumidors){
                        productor.signal();
                }
                lock.unlock();
                return tmp;
        }

}
```

2.-

```
// Client
```

```
class BufferBroadcastUnStub {

        MySocket s;

        BufferBroadcastUn() {
                s = new MySocket(localhost, 50000);
        }

        void putValue(Object v) {
                s.println("putValue");
                s.writeObject(v);
                s.readLine(); // sync
        }

        Object getValue(int id) {
                s.println("getValue");
                s.writeInt(id);
                return s.readObject(); // sync
        }
}

class Producer extends Thread {

        BufferBroadcastUnStub b;

        Producer(BufferBroadcastUnStub b) {
                this.b = b;
        }

        public void run() {
                while (true) {
                        System.out.println("about to produce...");
                        // v = ...
                        b.putValue(v);
                }
        }
}

class Consumer extends Thread {

        BufferBroadcastUnStub b;
        int id;

        Consumer(BufferBroadcastUnStub b, int id) {
                this.b = b;
                this.id = id;
        }

        public void run() {
                while (true) {
                        Object v = b.getValue(id);
                        System.out.println("about to consume...");
```

```
				// ... = v
			}
		}
}

class TestBufferBroadcastUnClient {

		final BufferBroadcastUnStub b = new BufferBroadcastUnStub();

		// producer
		new Producer(b).start();

		// consumers
		for (int i = 0; i < N; i++)
			new Consumer(b, i).start();
}

// Server
class BufferBroadcastUnSkel extends Thread {

		BufferBroadcastUn b;
		MySocket s;

		BufferBroadcastUnSkel(BufferBroadcastUn b, MySocket s) {
			this.b = b;
			this.s = s;
		}

		public void run() {
			while (true) {

				String op = s.readLine();

				if ("putValue".equals(op)) {
					b.putValue(s.readObject());
					s.println();
				} else if ("getValue".equals(op)) {
					s.writeObject(b.getValue(s.readInt()));
				} else {
					// error
				}
			}
		}
}

class TestBufferBroadcastUnServer {

		MyServerSocket ss = new MyServerSocket(50000);
		BufferBroadcastUn b = new BufferBroadcastUn();

		while (true) {
			MySocket s = ss.accept();
```

```
                    new BufferBroadcastUnSkel(b, s).start();
          }
}
```

3.-

```
- public void sendData(byte[] data, int data_off, int data_len) throws IOException {
          log.debug("%1$s->sendData(data.length=%2$d,data(0)=0x%3$x)", this, data_len,
data[data_off]);
          int n = 0;
          while (data_len > n) {
                    int len = data_len - n;
                    if (len > sndMSS) len = sndMSS;
                    TCPSegment segment = new TCPSegment();
                    segment.setSourcePort(localPort);
                    segment.setDestinationPort(remotePort);
                    segment.setFlags(TCPSegment.PSH);
                    segment.setData(data, data_off + n, len);
                    sendSegment(segment);
                    n += len;
          }
}
```

4.-

```
-                   state = LISTEN;

-                   try {
                              // wait some client to connect to me
                              while (acceptQueue.empty()) appCV.await();


-                             while (state != ESTABLISHED) appCV.await(); // wait SYN is received

-                                       ntcb.initActive(sourceAddr, rseg.getSourcePort(), ESTABLISHED);

-                                       appCV.signalAll();

-                                       state = ESTABLISHED;
                                        appCV.signalAll();
```

5.-

TCP provides a flow-control service to its applications to eliminate
the possibility of the sender overflowing the receiver's buffer.

Flow control is thus a speed-matching service, matching the
rate at which the sender is sending against the rate at which the
receiving application is reading.

Flow control is different from congestion control:

We suppose throughout this section that the TCP implementation is
such that the TCP receiver discards out-of-order segments.

TCP provides flow control by having the sender maintain a variable
called the receive window.

Informally, the receive window
is used to give the sender an idea of how much free buffer space is
available at the receiver.

Let's investigate the receive window in the context of a file
transfer. Suppose that Host A is sending a large file to Host B over
a TCP connection. Host B allocates a receive buffer to this
connection; denote its size by RcvBuffer. From time to
time, the application process in Host B reads from the buffer.

Define the following variables:

1.-LastByteRead: the number of the last byte in the data
stream read from the buffer by the application process in B
2.-LastByteRcvd: the number of the last byte in the data stream that
has arrived from the network and has been placed in the receive buffer
at B

Because TCP is not permitted to overflow the allocated buffer, we
must have:

LastByteRcvd - LastByteRead <= RcvBuffer

The receive window, denoted rwnd is set to the amount of
spare room in the buffer:

rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)

Host B tells Host A how much spare room it has
in the connection buffer by placing its current value of
rwnd in the receive window field of every segment
it sends to A.

Initially, Host B sets rwnd = RcvBuffer.

Host A in turn keeps track of two variables:

LastByteSent and LastByteAcked

LastByteSent - LastByteAcked: is the amount of
unacknowledged data that A has sent into the connection.

By keeping the amount of unacknowledged data less than the
value of rwnd, Host A is assured that it is not
overflowing the receive buffer at Host B.

Thus, Host A makes sure throughout the connection's life that

LastByteSent - LastByteAcked <= rwnd

Minor technical problem:

Suppose Host B's receive buffer becomes full so that

rwnd = 0

After advertising rwnd = 0 to Host A, also suppose
that B has nothing to send to A!!

Now consider what happens. As the
application process at B empties the buffer, TCP does not send new
segments with new rwnd values to Host A; indeed, TCP sends
a segment to Host A only if it:

- has data to send
- has an acknowledgment to send.

Therefore, Host A is never informed that some space has
opened up in Host B's receive buffer-Host A is blocked and can
transmit no more data!

To solve this problem, the TCP specification requires Host A to
continue to send segments with one data byte when B's receive window
is zero . These segments will be acknowledged by the receiver.
Eventually the buffer will begin to empty and the acknowledgments
will contain a nonzero rwnd value.