Computer Networks (CS425)

Instructor: Dr. Dheeraj Sanghi

Prev | Next | Index

Unix Socket Programming (Contd..)

Multiple Sockets

Suppose we have a process which has to handle multiple sockets. We cannot simply read from one of them if a request comes, because that will block while waiting on the request on that particular socket. In the meantime a request may come on any other socket. To handle this input/output multiplexing we could use different techniques :

1. **Busy waiting:** In this methodology we make all the operations on sockets non-blocking and handle them simultaneously by doing polling. For example, we could use the read() system call this way and read from all the sockets together. The disadvantage in this is that we waste a lot of CPU cycles. To make the system calls non-blocking we use:

   fcntl (s, f_setfl, fndelay);

2. **Asynchronous I/O:** Here we ask the Operating System to tell us whenever we are waiting for I/O on some sockets. The Operating System sends a signal whenever there is some I/O. When we receive a signal, we will have to check all sockets and then wait till the next signal comes. But there are two problems - first, the signals are expensive to catch and second, we would not be able to know if an input comes on a socket when we are doing I/O on another one. For Asynchronous I/O, we have a different set of commands (here we give the ones for UNIX with a VHD variant):

   signal(sigio, io_handler); fcntl(s, f_setown, getpid()); fcntl(s, f_setfl, fasync);

3. **Separate process for each I/O:** We could as well fork out 10 different child processes for 10 different sockets. These child processes are very light weight and have some communication between them. Now these processes waiting on each socket can have blocking system calls. This wastes a lot of memory, data structures and other resources.
4. **Select() system call:** We can use the select system call to instruct the Operating System to wait for any one of multiple events to occur and to wake up the process only if one of these events occur. This way we would know that the I/O request has come from which socket.

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout); void FD_CLR(int fd, fd_set *fdset); int FD_ISSET(int fd, fd_set *fdset); void FD_SET(int fd, fd_set *fdset); void FD_ZERO(fd_set *fdset);

The **select()** function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, **select()** blocks up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors. The nfds argument specifies the range of file descriptors to be tested. The **select()** function tests file descriptors in the range of 0 to nfds-1. **readfds, writefds and errorfds** arguments point to an object of type **fd_set**. **readfds** specifies the file descriptors to be checked for being ready to read. **writefds** specifies the file descriptors to be checked for being ready to write, **errorfds** specifies the file descriptors to be checked for error conditions pending.

On successful completion, the objects pointed to by the **readfds, writefds, and errorfds** arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. For each file descriptor less than nfds, the corresponding bit will be set on successful completion if it was set on input and the associated condition is true for that file descriptor. The timeout is an upper bound on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If the timeout is a null pointer, **select()** blocks until an event causes one of the masks to be returned with a valid (non-zero) value. If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, **select()** completes successfully and returns 0.

Reserved Ports

Port numbers from 1-1023 are reserved for the superuser and the rest of the ports starting from 1024 are for other users. But we have a finer division also which is as follows :

- 1 to 511 - these are assigned to the processes run by the superuser
- 512 to 1023 - they are used when we want to assign ports to some important user or process but want to show that this is a reserved superuser port
- 1024 to 5000 - they are system assigned random ports
- 5000 to FFFF - they are used to assign a port to user processes or sockets used by users

---

Some Topics in TCP

Acknowledgement Ambiguity Problem

There can be an ambiguous situation during the Retransmission time-out for a packet . Such a case is described below:
The events are :
1. A packet named 'X' is sent at time 't1' for the first time .
2. Timeout occours for 'X' and acknowledgement is not recieved .
3. So 'X' will be retransmitted at time 't2' .
4. The acknowledgment arrives at 't' .
In this case , we cannot be sure wether the acknowledgement recieved at 't' is for the packet 'X' sent at the time 't1' or 't2'. What should be our RTT sample value?

If we take ('t'-'t1') as the RTT sample :
t2-t1 (timeout period) is typically much greater than the average RTT . This implies that if we take t-t1 , then it will tend to increase the average RTT . If this happens for a large number of packets then the RTT will increase significantly. Now as the RTT increases , the timeout value increases and the damage caused by the above sequence of events increases . So this may cause the timeout to become very large unnecessarily .

If we take 't'-'t2' as the RTT sample :
Consider the case in which the acknowledgement for a packet takes a lot more time to arrive as compared to the timeout value . That is , the acknowledgement that arrives is for the packet which had been sent at the time 't1'.
That implies t-t2 is likely to be smaller and hence will tend to decrease the value of the RTT when included as a sample . A string of such events would cause the timeout to decrease . As the timeout decreases , the above sequence of events becomes more probable leading to even a further decrease in timeout . So if we consider t-t2, the timeout may become very small .

Since both the cases may lead to some problems , one possible solution is to discard the sample for which timeout occours . But this can't be done!!If the packet gets lost , the network is most probaaly congested The previous estimate of RTT is now meaningless as it was for an uncongested network and the characteristics of the network have changed Also new samples cant be found due to the above ambiguity .
So we simply adopt the policy of doubling the value of RTO on packet loss . When the congestion in the network subisdes , then we can start sampling afresh or we can go back to the state before the congestion occurred .
Note that this is a temporary increase in the value of RTO, as we have not increased the value of RTT. So, the present RTT value will help us to go back to the previous situation when it becomes normal.
This is called the **Acknowledgment Ambiguity Problem**.

Fast Restransmit

TCP may generate an immediate acknowledgment (a duplicate ACK) when an out- of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

This algorithm is one of the algorithms used by the TCP for the purpose of Congestion/Flow Control. Let us consider ,a sender has to send packets with sequence numbers from 1 to 10 (Please note ,in TCP the bytes are given sequence numbers, but for the sake of explanation the example is given). Suppose packet 4 got lost.

How will the sender know that it is lost ?

The sender must have received the cumulative acknowledgment for packet 3. Now, time-out for packet 4 occurs. On the receiver side, the packet 5 is received. As it is an out of sequence packet, the duplicate acknowledgment (thanks to it's cumulative nature) is not delayed and sent immediately. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. So, now the sender sees a duplicate ACK. But can it be sure that the packet 4 was lost ?

Well, no as of now. Various situations like duplication of packet 3 or ACK itself, delay of packet 4 or receipt of an out of sequence packet etc. might have resulted in a duplicate ACK.

So, what does it do? Wait for more!!! Yeah, it waits for more duplicate packets. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. This is called **Fast Retransmit**.

Flow Control

Both the sender and the receiver can specify the number of packets they are ready to send/receive. To implement this, the receiver advertises a Receive Window Size. Thus with every acknowledgment, the receiver sends the number of packets that it is willing to accept.

Note that the size of the window depends on the available space in the buffer on the receiver side. Thus, as the application keeps consuming the data, window size is incremented.

On the sender size, it can use the acknowledgment and the receiver's window size to calculate the sequence number up to which it is allowed to transmit. For ex. If the acknowledgment is for packet 3 and the window size is 7, the sender knows that the recipient has received data up to packet 3 and it can send packets of sequence number up to (7+3=10).

The problem with the above scheme is that it is too fast. Suppose, in the above example, the sender sends 7 packets together and the network is congested. So, some packets may be lost. The timer on the sender side goes off and now it again sends 7 packets together, thus increasing the congestion further more. It only escalates the magnitude of the problem.

---

back to top
Prev| Next | Index

lows.