UMBC CSEE

Advanced Operating Systems (CS-621)

Assignment-1

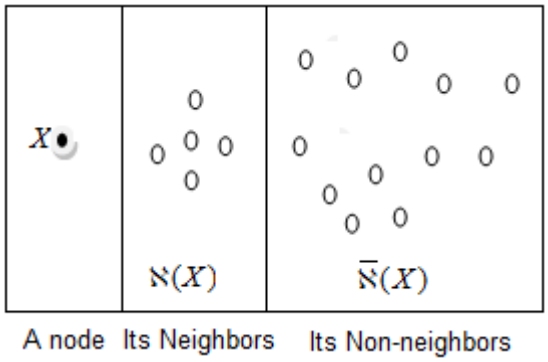
Sandipan Dey

2.8. Consider an unstructured overlay network in which each node randomly chooses **c** neighbors. If ***P*** and ***Q*** are both neighbors of ***R***, what is the probability that they are also neighbors of each other?

Solution

Let's assume that the underlying network has *n* nodes. Also, let's define the following set $\aleph(X) = \{Y \mid Y \neq X \ \wedge Y \ is \ a \ neighbor \ of \ X\}$, where both of *X* and *Y* are nodes in the underlying network (avoiding self-loops in the graph). Obviously the complement set $\overline{\aleph}(X) = \{Y \mid Y \neq X \ \wedge \ Y \ is \ NOT \ a \ neighbor \ of \ X\}$

By condition, $|\aleph(X)| = c, \ |\overline{\aleph}(X)| = n-1-c, \ \forall X$, with $c \leq n-1$. Hence, the entire network can be thought of in terms of union of 3 disjoint sets (the neighborhood relation induces 3 partitions; we assume the relation id not reflexive, i.e., $X \notin \aleph(X)$), $X, \ \aleph(X), \ \overline{\aleph}(X), \ \forall X$,as shown in the following figure, where we have $|X| + |\aleph(X)| + |\overline{\aleph}(X)| = 1 + c + n - 1 - c = n, \ \forall X$.



Assumptions

In case of overlay network, where every node has c randomly generated nodes as its neighbor,

- Neighborhood relation is not reflexive, i.e., a node has c randomly generated nodes which don't include itself (no self loops, the underlying graph is simple).
- The neighborhood relation is not symmetric. Since the neighbors for each and every node is generated randomly. Hence, node A has randomly generated node B and added in its neighbor list does not necessarily mean node B will also have A in its neighbor list.
- These neighbors are generated using some i.i.d. random variables, i.e., generation of neighbor lists for two different nodes are statistically independent.

Now, let's calculate the probability that an arbitrary node Y (≠ X) is a neighbor of a given node X.

$$\Pr(Y \in \aleph(X))$$

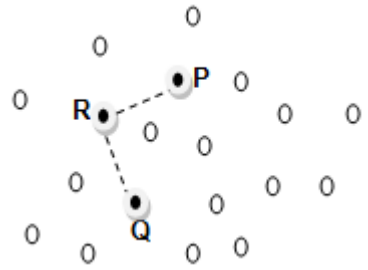
$$= \frac{\begin{pmatrix} \#\,ways\ that\ c-1\ other\ nodes\ can\ be\ selected\ from\ n-2\ nodes \\ when\ Y\ is\ already\ selected\ as\ a\ neighbor \end{pmatrix}}{\#\,ways\ that\ c\ nodes\ can\ be\ selected\ from\ n-1\ nodes} = \frac{\binom{n-2}{c-1}}{\binom{n-1}{c}}$$

$$= \frac{\frac{(n-2)!}{(c-1)!(n-c-1)!}}{\frac{(n-1)!}{c!(n-c-1)!}} = \frac{c}{n-1} = \frac{|\aleph(X)|}{|\aleph(X)| + |\overline{\aleph}(X)|}$$

and also we know (from Pascal's triangle)

$$\underset{\substack{\#\,ways\ to\ select\ c\ neighbors \\ of\ X\ \ from\ n-1\ nodes}}{\binom{n-1}{c}} = \underset{\substack{\#\,ways\ to\ select\ c\ neighbors \\ of\ X\ \ from\ n-1\ nodes \\ when\ Y \in \aleph(X)}}{\binom{n-2}{c-1}} + \underset{\substack{\#\,ways\ to\ select\ c\ neighbors \\ of\ X\ \ from\ n-1\ nodes \\ when\ Y \notin \aleph(X)}}{\binom{n-2}{c}}$$

Now, with all the above results, let's proceed to solve the given problem:
Given, $P, Q \in \aleph(R)$



But, since the *c* neighbors for any node are selected randomly independent of other nodes' selection of neighbors, we have, $\Pr(P \in \aleph(Q) \mid P, Q \in \aleph(R)) = \Pr(P \in \aleph(Q))$. Now, by our earlier discussion, we know, $\Pr(P \in \aleph(Q)) = \Pr(Q \in \aleph(P)) = \frac{c}{n-1}$, so that

$$\Pr(P \in \aleph(Q)) + \Pr(Q \in \aleph(P)) = \frac{2c}{n-1}$$

$$\Pr(P \in \aleph(Q) \ \wedge \ Q \in \aleph(P)) = \Pr(P \in \aleph(Q)).\Pr(Q \in \aleph(P)) = \left(\frac{c}{n-1}\right)^2$$, by independence.

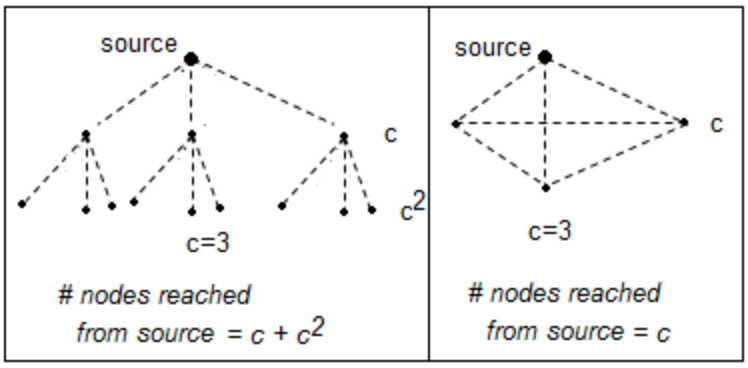
$$\Pr(P \in \aleph(Q) \ \vee \ Q \in \aleph(P))$$
$$= \Pr(P \in \aleph(Q)) + \Pr(Q \in \aleph(P)) - \Pr(P \in \aleph(Q) \ \wedge \ Q \in \aleph(P))$$
$$= \frac{2c}{n-1} - \left(\frac{c}{n-1}\right)^2$$

2.9. Consider again an unstructured overlay network in which every node randomly chooses c neighbors. To search for a file, a node floods a request to its neighbors and requests those to flood the request once more. How many nodes will be reached?

Solution

If # nodes reached = *n,* (**excluding the source node**), it's easy to see that $c \le n \le c(c+1)$.
More precisely, if # nodes in the network = *N*, so that there are *N - 1* other nodes apart from this node, we have, $\max(c, N-1) \le n \le \min(c(c+1), N-1)$.

The exact result depends upon how the neighbor nodes were chosen by each of the nodes, whether they are overlapped or not. If the neighbors chosen by different nodes are disjoint, we have maximum number of nodes reached, i.e., *c(c+1)*, in case of the overlay tree. On the other extreme, if we have an (almost) fully connected overlay graph (i.e., *c* = *O(n)*) , then number of nodes reached = *c*.



But in general, c << n, hence number of nodes reached = $O(c^2)$.

3.1. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in a cache in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it's single-threaded? If it's multi-threaded?

Solution

Average service time for a request $= \frac{2}{3} \times 15 + \frac{1}{3} \times (15 + 75) = 10 + 30 = 40 m\sec$

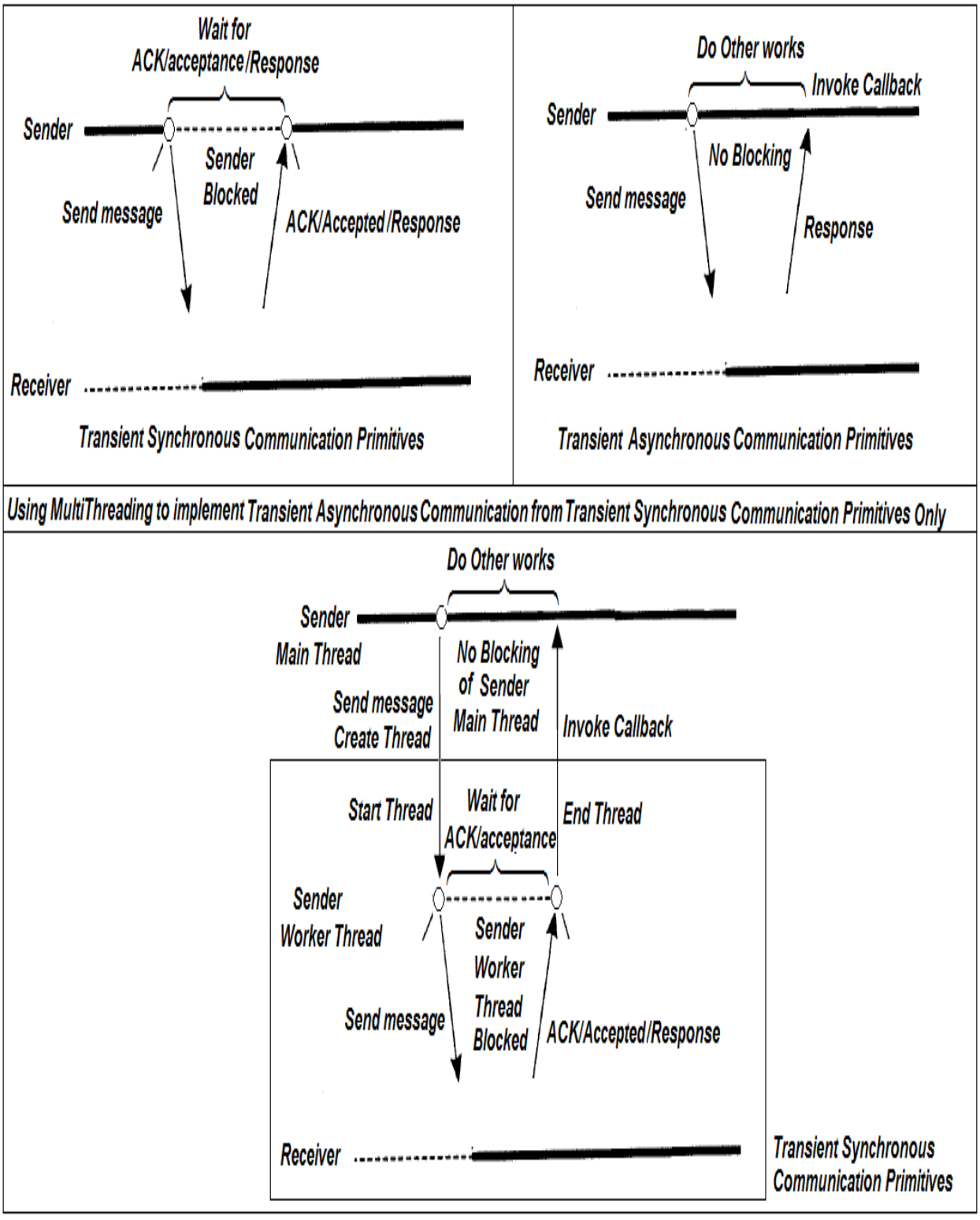
Hence, requests/sec the server can handle $= \frac{1}{40 \times 10^{-3}} = \frac{1000}{40} = 25$

4.13. Suppose that you could make use of only transient synchronous communication primitives. How would you implement primitives for transient *asynchronous* communication?

Solution

The only difference between the asynchronous and synchronous mode of communication is that in the first mode the sender is allowed to continue other works as soon as the message is submitted for sending, but in the later mode the sender has to wait (till receiver receives a message or till the message delivery has taken place or even till the receiver responds, depending upon the point of synchronism).

Now, we have only the transient synchronous communication primitives at our disposal. In order to implement transient asynchronous communication primitives we can use multithreading. We can have a sender main and another sender worker thread, and delegate (move) the synchronous blocking wait inside the sender worker thread while the main thread can continue with other works immediately after submitting the message (buffering the message in the worker thread's message queue) for sending. On completion of receiving response from the receiver, the worker thread can invoke a callback to notify the main thread. The entire lifetime of the worker thread will be spent in waiting from the response from the receiver, whether main thread can do its own work meanwhile. The approach is shown in the following figure.



Sample implementation (in java) may look like the following:

```
// Sender Main Thread
class SenderMain
{
    private senderWorker;
    //...

    public void sendMessage(MSG msg)
    {
        senderWorker = new SenderWorker(this, msg);
        senderWorker.start();
    }

    // Callback
    public synchronized void onReceive()
    {
        // ... update
    }

    public static void main(String[] args)
    {
        SenderMain main = new SenderMain();
        MSG msg = new MSG();
        // ... prepare message to be sent here
        main.sendMessage(msg);
        // ...
    }
}
// Sender Worker Thread
class SenderWorker extends Thread
{
    private MSG msg;
    private SenderMain main;
    //...

    public void SenderWorker(SenderMain main, MSG msg)
    {
        this.main = main;
        this.msg = msg;
    }

    public void run()
    {
        // Use transient synchronous communication primitives
        // wait here in loop
        // on response from the receiver invoke the callback
    }
}

class MSG
{
    // ...
}
```