HW-2

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 he 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 is single threaded? If it is multi threaded?

A:
In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is ((2/3 × 15) + (1/3 × 90)). Thus the mean request takes 40 msec and the server can do 25 per second.
For a multi-threaded server, all the waiting for the disk is overlapped and the cache hits takes 15msec as in single-threaded but the cache misses also takes 15msec unlike single-threaded because during the 75 msec additional time the thread sleeps. Thus the weighted average is ((2/3 × 15) + (1/3 × 15)). So every request takes 15 msec, and the server can handle 66 2/3 requests per second.

2: Would it make sense to limit the number of threads in a server process?

A: Yes, the number of threads in a server process should have limits for the following reasons. Threads use their private stack which requires memory. Hence, having too many threads would result in over utilization of memory which reduces the performance. Another reason is that , independent threads tends to operate in chaotic manner. This may result in many page faults and thus I/O. Thus, having many threads may result in performance degradation from page thrashing.

3: Constructing a concurrent server by spawning a process has some advantages and disadvantages compared to multithreaded servers. Mention a few.

A: Spawning a process is a technique where OS creates a child process by the request of another process. It is a relatively costly operation as compared to multi-threaded servers. Also, using threads is much cheaper if the processes do need to communicate, rather than kernel implementing each communication. Another advantage of spawning a process is that separate processes are protected against each other.

4: Is a server that maintains a TCP/IP connection to a client stateful or stateless? Justify your answer.

A. Stateless protocols does not require the server to retain the session information or session details whereas stateful protocol saves the status and session information. In TCP/IP connections the server does not maintain the session information, hence it can be concluded that it follows stateless protocol.

5: Describe how connection-less communication between a client and a server proceeds when using sockets.

A: User Datagram Protocol (UDP) is used in connection-less communication between a client and server instead of TCP/IP. Both the client and the server create a socket, but only the server binds the socket to a local endpoint.

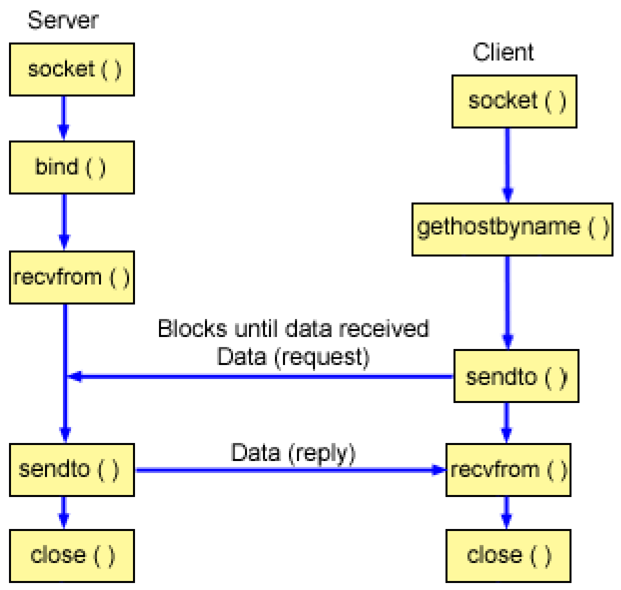

Fig. *Client/server relationship of the socket APIs used in the examples for a connectionless socket design.*

Flow of events for server:
1. The socket() API returns a socket descriptor, which represents an endpoint.
2. After the socket descriptor is created, a bind() API gets a unique name for the socket.
3. The server uses the recvfrom() API to receive the data. The recvfrom() API waits indefinitely for data to arrive.
4. The sendto() API echoes the data back to the client.
5. The close() API ends any open socket descriptors.

Flow of events for client:
1. The socket() API returns a socket descriptor, which represents an endpoint.
2. If the server string that was passed into the inet_pton() API was not a valid address string, then it is assumed to be the host name of the server. In that case, use the getaddrinfo() API to retrieve the IP address of the server.
3. Use the sendto() API to send the data to the server.
4. Use the recvfrom() API to receive the data from the server.
5. The close() API ends any open socket descriptors.

6: Does it make sense to implement persistent asynchronous communication by means of RPCs ?

A: Yes, it does makes sense because the methodology of RPC communication is quite similar to persistent asynchronous communication. But it is necessary for the RPC to be on a hop-to-hop basis in which a process managing a queue passes a message to a next queue manager by means of RPC. The message is stored by the store manager. The manager of the client queue offers a implementation for proxy of the interface to the manager of the server queue and receives success or failure of each operation.

7: With persistent communication, a receiver generally has its own local buffer where messages can be stored when the receiver is not executing. To create such a buffer, we may need to specify its size. Give an argument why this is preferable, as well as one against specification of the size.

A: Buffer implementation becomes easier if the user specifies the size of the buffer. However, in this method the messages might be lost if the buffer is full. This might happen when the receiver is not in execution for a long time and hence the buffer gets filled up and any further message gets lost. On the other hand, if the system manages the buffer size ,then the system requires to do much more work as it would have to shrink or increase the buffer size as per requirement which eventually may affect the performance in a negative way but no message would be lost.

8: When searching for files in an unstructured peer-to-peer system, it may help to restrict the search to nodes that have files similar to yours. Explain how gossiping can help to find those nodes.

A: In an unstructured peer-to-peer system, during gossiping, if each nodes exchange the membership information to every other node, then each node of the system will eventually get to know about all other nodes in the system. If a new node is discovered then each time it is evaluated using the semantic proximity. The semantically nearest nodes are then selected for submitting a search query. Thus it may help to restrict the search to nodes that have similar files.