1.

a)

if the situation is happened where the single thread model used, in case of the cache hits, take 15 ms; in case of cache misses, it would take 15+75 = 90 msec.
The calculation of weighted average can be done via 2/3 x 15 + 1 /3 * 90 = 40ms. Which means that average/mean time of a common request takes 40 ms. Thus, this server can handle 25 requests per second.

b)
In case of multithreaded server, all the waiting for the disk operation is overlapped, and handled by multithreads, thus we could consider that each request takes 15 msec. After calculations, performance of requests per second be done by server is 66 2/3.

2.

It does make sense. Although thread can be considered as a light weight process that require less resource against process, it still need resources such as private stack memory space, I/O requests etc. Setting up too much unnecessary threads would consume too much memory in server thus may result server cannot work properly. Also, in an OS, independent threads' virtual memory pages are not stable, too many threads can lead to degradation of performance because of page trashing.

3.

Consider to create a separate process Q as a file server that handles remote requests for File F. In implementation, Process Q can offer Process P the same interface to F like the situation in same machine, for instance, It can be implemented in the form of a proxy.

4.

shared across a multithreaded process:

Global values

Heap memory

5.

Multi-process (1) vs. Single-process, multi-threaded (2):

Impact of segmentation faults:

In (2), if one thread causes segmentation fault, the whole application crashes.

In (1), if processes considered as modules in this system, they have different memory regions and thus only the module that cause segmentation fault will crash.

Communication:

In (2), Threads share same heap, same global variables, communication is much easier and guaranteed.

In (1), it is required using a method that is out of execution context (either sockets way or IPC) to communication with each other, thus harder to implement and maintain.

Sharing memory:

In (2), the whole memory is shared by all threads, so you can directly send message objects. Also, it is possible that a thread can use any part of others with correct offsets.

In (1), you need to use 'Shared Memory' such as IPC between processes.

Pointer usage between modules:

In (2), you can use pointers in your message objects. The ownership of heap objects (accessed by pointers in the messages) can be transferred to the receiving module.

In (1), you need to manually manage the memory (with custom malloc/free functions) in the 'Shared Memory' region such as IPC.

Module management:

In (2), you are managing just one process.

In (1), you need to manage a pool of processes each representing one module.

System resource consuming:

To complete same task, the multiple threads would consume less resource than multiple process.

Performance:

It depends on much sectors but in general, a multiple process would have better performance in massive float calculations. (especially in windows) However, processes are heavier to start.

6.

As I said in problem 5. It is depending on multiple sectors.

For CPU bound tasks where you have more than one core in your processor you can divide your work on each of your processor core. If you have two cores, split the work on two threads. This way you have to threads working at full speed. However, threads are really expensive to create, so you need a pretty big workload to overcome the initial cost of creating the threads. Also, it depends on OS. Some operation system treated a single process with multiple user threads as single kernel thread. It would not improve performance if you do single massive calculation.

7.

Preemptive scheduling will interrupt a thread that has not blocked or yielded after a certain amount of time has passed (amount of time depends on the CPU scheduling algorithm). Non-preemptive scheduling will not interrupt the thread. The OS will wait until the thread blocks or yields.

8.

User-level threads are the threads that the OS is not aware of. They exist entirely within a process, and are scheduled to run within that process's time slices.

The OS is aware of kernel-level threads. Kernel threads are scheduled by the OS's scheduling algorithm, and require a "lightweight" context switch to switch between (for example, registers, PC, and SP must be changed, but the memory context remains the same among kernel threads in the same process).

User-level threads are much faster to switch between, because there is no context switch and did not require any syscall which would interrupt kernel; furthermore, a problem-domain-dependent algorithm can be used to schedule among them. CPU-bound tasks with interdependent computations, or a task that will switch among threads often, might best be handled by user-level threads.

9.

Process consumes more resources. OS need more resource to start a process and take more care of a process.

A thread can be considered a “light-weight” process, which is sharing global variables, heap space with other threads inside a process. Usually, a process contains a primary thread.

Threads inside a process are bound to this process, it’s much easier to communicate between threads against communication between processes.

Process start up with its own address space, text, data in memory. Thread could be fired up with a single sequential execution steam within a process.

10.

One of the scalabilities problems that the protocol X suffers:

Large amount of bandwidth needed.

One possible solution to tackled this particular issue is using compression techniques.

Let’s say a message packet are considered to consist of several parts in a fixed manner, for example, structured data form, like an identifier part and a variable part. In many cases, multiple messages will have the same identifier. Also, in these particular cases, they will often contain similar data. A compression approach can be made that only send the differences between messages which has the same identifier. Both the sending and receiving side setup and maintain a local cache. In this cache pool, the entries can be looked up as identifier of a message. Thus, when a message is sent, it is first looked up in the local cache, and If found a previous message with the same identifier but possibly with different data need to send, it could use differential encoding to send only the differences between the current message and previous message. At the receiving side, the message is also looked up in the local cache after decoding through the differences. If cache miss, then using standard compression technology such zip. This solution can generally improve bandwidth usage performance.

Another scalability problem is that “X” still requires having a display running (the display generally needs to synchronize a lot).

One possible solution is to keep the software at the display very simple is to let the entire processing take place at the application side. Effectively, this means that the entire bitmap are sent over the network to the display, where they are immediately transferred to the local frame buffer.