Assignment 3

Georgia Koloniari
Salteas-Kalogeras Panagiotis

Problem 9:

We discern two approaches for designing a multithreaded server that supports multiple protocols by using sockets.

The first approach is based on the use of a single superserver that listens to a well-known endpoint. During the initialization the server calls *socket* and *bind* to bind the endpoint (port) where it would accept client requests. The superserver then calls *listen* and *accept* and waits for incoming client requests. When a client wants to communicate with the server by using a specific protocol, it sends a request to the well-known port of the superserver specifying the protocol that it wishes to use in the rest of their communication. When the server receives the request it creates a new thread that will be responsible to communicate with the client by using the specified protocol. The new thread also calls *socket* and *bind* to bind to a new endpoint that will be used for its communication with the requesting client. The thread answers to the client from the new endpoint and their communication can now begin. After the client is served the thread is destroyed. Thus, this approach actually creates one thread per client that is destroyed after the client is served.

The second approach we could use for the design of the multithreaded server uses a different thread for each of the different protocol that the server supports. We once again use the notion of the superserver that binds and listens to well-known port for incoming clients request. However in this approach, during the initialization of the server it creates a thread for each one of the different protocols it supports. Each thread binds to a different point which it eventually will use to communicate with the clients. A client sends a request to the superserver specifying the protocol it wishes to use. The server replies with the endpoint of the corresponding thread responsible for the specified protocol. Each thread now has a queue for clients that wait to be served while the thread is occupied with another client. The threads are obviously not destroyed after serving a client, and they continue working with the next client in their queue. In this approach, the number of threads running is fixed and does not depend on the number of clients.

Problem 12:

The main design issue we have to consider when creating an object adapter for persistent objects is the invoking policy we will follow. We can choose to create the object every time a client requests it, however since we are talking about persistent objects the natural solution is to create all the objects at the initialization of the server.

A persistent object is an object that continues to exist even if it is not currently contained in the server's address space. The object's stage can be stored at secondary storage and a new server can read it and manage the object. Thus the server will create the objects at the beginning and associate one thread per object. Whenever an invocation comes for an object the adapter responsible for implementing the activation policy will pass the client request to the corresponding thread. If the thread is busy the client will be queued in the thread's queue. This approach also improves the server's performance since we do not have to create each object for every request and lose time while reading its status, or storing its state on disk before destroying.

When designing the adapter we also need to ensure that we handle concurrent attempts of accessing the same data. By using a thread for each object as we proposed above, this problem is also solved without any further effort. However we are not interested in object specific code since the adapter is independent of the particular objects it manages and only depends on their activation policy. This is the only way we can construct generic object adapters and conceptually place them in the middleware.

Problem 17:

In UNIX systems we can achieve strong mobility by allowing a process to fork a child-process on a remote machine. Strong mobility is obtained when we transfer not only the code but the execution segment of a process as well. The fork procedure creates an exact copy of the parent procedure by copying the data space, the heap and the stack to the child procedure. Not only code and initial data but execution segment as well are copied and thus we have the strong mobility scenario. Since an exact copy is created, the above procedure is known as remote cloning. The cloned procedure is executed in parallel with the original process only in a different machine.