Iterated Connectionless Server Algorithm

1. Check a socket an bind to the address for the service being offered
2. Repeatedly read the next request from a client, create a response, and send a reply back to the client according to the application protocol decided upon.

Notes: In a connectionless server we cannot use the “connect” option described earlier. It must use an unconnected socket:

\[
\text{retcode} = \text{sendto}(s, \text{message}, \text{len}, \text{flags}, \text{toaddr}, \text{toaddrlen})
\]

How does a server determine address of a client?

\[
\text{retcode} = \text{recvfrom}(s, \text{buf}, \text{len}, \text{flags}, \text{from}, \text{fromlen})
\]

Concurrency

There is a need to provide faster response times to multiple clients. Response time is improved under the following situations:

- Forming a response requires significant I/O.
- Processing time can vary dramatically between requests
- Server executes on a computer with multiple processors

Concurrent connectionless server algorithm

**Master 1:** Create a socket and *bind* to the address. Leave socket unconnected.

**Master 2:** Repeatedly call *recvfrom* to get next request from client and create a new slave thread (possibly in a new process) to handle the response.

**Slave 1:** Begin with a specific request from the master (see Master 2 above) as well as access to the socket (i.e., receive access to socket).

**Slave 2:** Form a reply according to the application protocol and send back to client using *sendto*

**Slave 3:** exit, i.e., slave terminates after handling one request.
Concurrent connection-oriented server algorithm

*Master 1:* Create a socket and *bind* to the address. Leave socket unconnected.

*Master 2:* Leave socket in passive mode, making it ready for use by a server.

*Master 3:* Repeatedly call *accept* to receive the next request from a client, and create a new slave thread or process to handle the response.

*Slave 1:* Begin with a connection passed from the master (i.e., a socket for the connection).

*Slave 2:* Interact with the client using the connection: use *read* & *send*.

*Slave 3:* Close the connection and exit, i.e., slave terminates after handling all requests from one client.

When to use each server type

**Iterative/Concurrent:** Iterative servers are easier to design, implement, and maintain, but concurrent servers can provide quicker response to requests. Use an iterative implementation if request-processing time is short and an iterative solution produces response times that are sufficiently fast for the application.

**Real vs. apparent concurrency:** A server that has a single thread of execution depends on asynchronous I/O to allow it to manage multiple connections. A multi-threaded implementation, whether multiple, singly-threaded processes or multiple threads within a single process, allows the OS to provide concurrency automatically. Use a single-thread solution if the cost of creating or switching among threads is high and the server must share or exchange data among connections. Use a multi-threaded solution, if the server must share or exchange data among connections and the cost of using threads is low. Use a multi-process solution if each slave can operate in isolation or to achieve maximal concurrency (e.g., on a multiprocessor).

**Connection-oriented vs. connectionless:** Because connection-oriented access means using TCP, it implies reliable delivery; connectionless transport (UDP) implies unreliable delivery. Only use connectionless transport of the application protocol handles reliability or each client accesses its server on a local area network that exhibits extremely low loss and no packet reordering. Use connection-oriented transport whenever a wide area network separates the client and server. Never move a connectionless client and server to a wide area environment without checking to see if the application protocol handles reliability problems.

Summary of server types

**Iterative, connectionless server:** The most common form of connectionless server, used especially for services that require a trivial amount of processing for each request. Iterative servers are often stateless, making them easier to understand and less susceptible to failures.
**Iterative, connection-oriented server**: Less common. Used for services that require a trivial amount of processing for each request, but for which reliable transport is necessary. Because the overhead associated with establishing and terminating connections can be high, the average response time can be non-trivial.

**Concurrent, connectionless server**: An uncommon type in which the server creates a new thread or process to handle each request. On many systems, the added cost of thread or process creation dominates the added efficiency gained from concurrency. To justify concurrency, either the time required to create a new thread/process must be significantly less than the time required to compute a response or concurrent requests must be able to use many I/O devices simultaneously.

**Concurrent, connection-oriented server**: It is the most general type of server because it offers reliable transport (i.e., it can be used across a wide area Internet) as well as the ability to handle multiple requests concurrently. Two basic implementations exist: the most common implementation uses concurrent processes or concurrent threads within the same process to handle connections; a far less common implementation relies on single thread and asynchronous I/O to handle multiple connections.

In a concurrent process implementation, the master server thread creates a slave process to handle each connection. Using multiple processes makes it easy to execute a separately compiled program for each connection instead of writing all the code in a single, large server program.

In the single-thread implementation, one thread of execution manages multiple connections. It achieves apparent concurrency by using asynchronous I/O. The server repeatedly waits for I/O on any of the connections it has open and handles each request. Because a single thread handles all connections, it can share data among them. However, because it has only one thread, the server cannot handle requests faster than an iterative server, even on a computer that has multiple processors. The application must require data sharing or the processing time for each request must be short to justify this server implementation.
The following figures illustrate some of the types of servers described in the previous pages.

**Figure 1**: Process structure for an iterative, connectionless server. A single thread of execution communicates with many clients using a single socket.

**Figure 2**: Process structure for an iterative, connection-oriented server. A single thread of execution waits at a well-known port for a connection and then communicates with the client over that connection. Connection initiation is over another port.
Figure 3: Process structure for an iterative, connection-oriented server. A single thread of execution waits at a well-known port for a connection and then communicates with the client over that connection. Connection initiation is over another port.

Figure 4: Process structure of a connection-oriented server that achieves concurrency with a single thread. The thread manages multiple sockets.