Server algorithms and their design

- many ways that a client/server can be designed
- each different algorithm has various benefits and problems
  - are able to classify these algorithms by looking at the various server differences
- the main client/server algorithms can be classified:
  - iterative connectionless server
  - iterative connection oriented server
  - concurrent connectionless server
  - concurrent connection-oriented server

The basic server algorithm

- overview of a server

- conceptually each server follows a simple algorithm:

```
it creates a socket
binds the socket to a well known port
loop
   accept the next client
       request from this port
   serve this request
   formulate a reply
   send the reply to client
end
```

Problems with the simple server?

- unfortunately this is only good enough for simple applications
- consider a service requiring considerable time to handle each request
  - example suppose an ftp client server were implemented like this!
  - one user requests a huge file
  - moments later another user might wish to transfer a small file
- the second user might has to wait a considerable time **just** to transfer a small file
- the second user **blocks** until the first user has finished with the server.
- thus servers are seldom built like this

Servers

- iterative server
  - used to describe a server implementation that processes one request at a time
- concurrent server
  - used to describe a server that handles multiple requests at one time
- from a clients perspective
  - the server appears to communicate with multiple clients concurrently.
- *The term concurrent server refers to whether the server handles multiple requests concurrently, not to whether the underlying implementation uses multiple concurrent processes*

Conclusions

- concurrent servers are more difficult to design and build
  - the resulting code is more complex
  - difficult to modify

- however most programmers choose concurrent server implementations as iterative servers
  - cause unnecessary delays in distributed applications
  - may be a performance bottleneck that effects many client applications

- *Iterative server implementations, which are easier to build and understand, may result in poor performance because they make clients wait for service. Whereas in contrast, concurrent server implementations, which are more difficult to build, yield better performance.*

Iterative connection oriented servers and connectionless servers

- choice of transport protocol dictates choise of server
  - TCP provides a *connection-oriented*
  - UDP provides a *connectionless* service
- servers that use TCP are, by definition, *connection oriented*
- those that use UDP are *connectionless servers*
- should also examine the application protocol
  - because an application protocol designed to use a connection oriented protocol might perform incorrectly or inefficiently when using a connectionless protocol
  - why?

```
create a socket and bind
   to the well known address
   of the service being offered.

put socket into passive mode, making it
   ready for use by a server

loop
   accept the next connection
       request from the socket,
   and obtain a new socket
   for the connection

   repeat
      read a request from
          the client,
      generate a reply,
          send the reply
```

```
            back to the client
    until finished with the client ;
    close connection
end
```

Comments on algorithm

- a less common server
  - used for services that require a trivial amount of processing
  - it might incur a high overhead in establishing and terminating connections

Implementation notes

- `getportbyname`
  - use this function to map a particular service onto a well known port number
  - need to ensure that our application port number does not conflict with someone else application
  - in UNIX all port numbers are defined in `/etc/services`
- `bind`
  - use this function to bind a socket to a port number and the servers IP address
- `listen`
  - the server uses this function to place a socket into passive mode
  - it also takes an argument which specifies the length of request queue for this socket

Implementation notes

- `accept`
  - use this function to obtain the next incoming connection request
  - returns the new descriptor of a socket which can then be used by `read` and `write`

- `close`
  - to close the connection with the client the server uses `close`.

Iterative connectionless server

- iterative servers work best for services that have a low request processing time
  - the connection oriented protocol TCP has a high connection and disconnection overhead
  - thus most iterative servers use connectionless protocol such as UDP

-

```
create a socket and bind
   to a well known address
   for which a service is
   being offered
loop
   read next request from client
   process the request
```

```
    send reply back to client
end
```


Iterative connectionless server

- the most common form of connectionless server, used especially for service that require a trivial amount of processing for each request
  - iterative servers are often stateless, making them easier to understand and less vulnerable to failures

Implementation notes

- `sendto`
  - use this function to send data across a connectionless socket

- `recvfrom`
  - use this function to receive data from a connectionless socket

Concurrent connection oriented servers and connectionless servers

- primary reason for introducing concurrency
  - provide faster response time to multiple clients

- works well when
  - forming a response requires significant input output
  - the processing time required varies dramatically

```
create a socket and bind
    to the well known address
    for the service being offered

leave the socket unconnected

loop
   call recvfrom to obtain the
        next client request
   if (fork() == 0) {
       /* child process */
       do whatever the
          client request says
       form a reply and send
           it to client
       (use sendto)
       exit
   } else {
```

```
        /* must be the parent */
    }
end
```

Comments on the algorithm

- an uncommon type in which the server creates a new process to handle each request
  - note time to create a process may dominate the added efficiency gained from concurrency

Concurrent connection oriented server

- the most general type of server because it offers reliable transport as well as the ability to handle multiple requests at the same time
  - note that connection oriented servers implement concurrency among connections rather than individual requests

```
create a socket and bind
   it to the well known address
   for the service being offered

place socket into passive mode
   making it ready for use by
   the server

loop
   call accept to receive the
   next request from a client
   if (fork() == 0) {
       /* must be the child */
       repeat
          read request from client
          do whatever the client
             request says
          form a reply and send
```

```
            it to client
        until client wishes to quit
        close connection
        exit
    } else {
        /* must be the parent */
    }
end
```

When to use each server type

- iterative vs concurrent
  - iterative easier to design, implement and maintain
  - concurrent can provide quicker response to requests

- use iterative implementation if
  - request processing time is short
  - and the code produces fast responses times

- Connection oriented vs connectionless
  - connection oriented access means using TCP
    - implies reliable delivery
  - because connectionless transport means using UDP
    - it implies unreliable delivery

Conclusion

- only use connectionless transport if the application protocol handles reliability
  - or the local area network exhibits:
    - low packet loss
    - no packet reordering (very few do)

- use connection oriented transport whenever
  - a wide area network separates client and server

- never move a connectionless client and server to a wide area network
  - without checking to see if the application protocol handles the reliability problems