A Secure Pre-threaded and Pre-forked Unix Client-Server Design for Efficient Handling of Multiple Clients

Lokender Tiwari¹ and Vijay Gulashan Pandey²
¹U.N.S Institute of Engineering & Technology, ²V.B.S Purvanchal University, Jaunpur
¹loke1602@gmail.com and ²gulashan2001@gmail.com

ABSTRACT
A Server that can handle only one client at a time is not very effective and useful. Servers typically need to be able to handle multiple clients simultaneously. Three different general strategies are used for handling: multiplexing, forking and threading with their advantages and drawbacks. Basically servers fall into two category iterative server and concurrent server. In iterative server cannot process a pending client until it has completely serviced the current client. Concurrent server use concept of forking to spawn a child process to every client. There are also two modified design of concurrent servers. Pre-forking and Pre-threading. Pre-forking or process pre-allocation is used to control delay, and maintain high throughput in concurrent servers by lowering the server delay caused due to cost of creating process each time request arrives as compared to general forking. But in a multi tasking environment, pre-forked processes run as daemon processes, then they pose to security and reliability risks. When an attacker compromises one of the daemon processes, he can use this process to try to infect other processes more effectively because daemon processes never die. To avoid this vulnerability we use a performance and security pattern for resource pooling. Older processes are periodically evicted from the resource pool and new processes are spawned in their place. This pattern therefore mitigates the trade-off between performance and security. While in pre-threading one thread from the thread pool handles each client. Threads share memory space of parent process and hence have fast context switching time and resources used in this case are lesser than a multiprocess server. One big drawback of a pre-threaded server if a thread crashes, it bring down the entire server. Process wins in stability while Threads win in fastest context switching time. Thus, in this paper with these two individual advantages we combined the two server architectures and proposed a hybrid server architecture to achieve the speed of pre-threading and stability of pre-forking along with secure Pre-forking to prevent server from malicious attacker with improved performance and security that efficiently handle multiple clients simultaneously.

KEYWORDS
Iterative Server, Concurrent Server, Pre-threading, Pre-forking, Secure Server, Unix Server, Client-Server Model, Secure Pre-forking.

1. INTRODUCTION
This paper is designed with the aim of defining a performance and security enhanced hybrid pre-threaded and pre-forked client-server design for efficient handling of multiple clients. This paper includes various client-server alternatives and a comparative analysis of all alternatives and a pattern for performance and secured server.

2. CLIENT-SERVER ARCHITECTURE
Client-Server interaction is the basis of computer communication. One side must start execution and wait (indefinitely) for the other side to connect.

Fig1: Client-Server Model

Applications that initiates communication is called clients, and a server is one that wait for incoming communication request from clients.

3. SOCKETS-THE SOCKET INTERFACE
A socket is an abstraction for network communication, just as a file is an abstraction for file system communication. The interface that resulted from the task that is assigned by Advanced Research Group Projects Agency (ARPA) to a group at the University of California, Berkeley, to port TCP/IP software to the UNIX operating system become known as the socket interface sometimes Berkeley socket interface. The Berkeley socket interface extends the concept of a file descriptor to that of socket descriptor. A socket structure contains information such as socket type port being used by socket, local address, and the remote address and port that will receive communication from the socket.

Sockets can be used in two different ways, once we create a socket it can wait for the connection, or it can initiate a connection to another remote host or its own local host. A socket used by client program to initiate a connection to the server called active socket. While the socket that acts as a server and wait for the connection is known as passive client.[1]

4. TCP/IP CLIENT-SERVER SOCKET INTERFACE
Both client and server begin with a call to socket, returning a socket descriptor. Clients then call connect, while server call bind, listen, and accept and bind it to a address and port told that we're ready to receive connection requests. After creating
an active connection both client and server exchange data.[1][2]

![Diagram](image)

**Fig2: TCP/IP Client-Server Socket Programming Overview**

Both client and server begin with a call to socket, returning a socket descriptor. Clients then call connect, while server call bind, listen, and accept and bind it to a address and port told that we're ready to receive connection requests. After creating an active connection both client and server exchange data.[1][2]

### 5. TYPE OF SERVERS

On the basis of handling of multiple client we can mainly divide servers into two categories, Iterative server and Concurrent server. After further modification to concurrent servers we get two new design alternatives, Pre-forked and Pre-threaded concurrent servers.[1]

### 6. ITERATIVE SERVER MODEL

Iterative server can handle only one client at a time and cannot

![Diagram](image)

**Fig3: Iterative Server**

process pending client until it completely serviced the current client.

### 7. CONCURRENT SERVER MODEL

Concurrent servers handle multiple clients at a time. It uses the concept of forking and threading to spawn a child process for each client. We further modified the concurrent server model by using concept of pre-threading and pre-forking.

- **Pre-forking**- Pre-forking involves the forking of parent process or server process into number of child server processes, when it starts and create a pool of all these child server processes. So that one process from the available pool handles each client request.

- **Pre-threading**- Similar to pre-forking in pre-threading when server starts it creates a pool of available threads, so that each thread handles each clients.[1][2]

### 8. CONSIDERATION AND ASSUMPTIONS

Various issues regarding to pre-threaded and pre-forked server that is considered and analyzed in this paper are, What if there are not enough processes or threads in the pool? What if there are too many processes or threads in the pool? and How can the parent and its children or threads synchronize with each other.

We considered the process control CPU time (The time required to service a fixed number of client requests) for iterative server is zero and assume it our baseline. Process control CPU time in this paper means difference from the baseline for a given design.

Server timing were calculated by running the client on two different hosts on the same subnet as the server. Both clients spawned five children to create five simultaneous connections to the server maximum of 10 simultaneous connections for all tests. For pre-forked and pre-threaded server created 15 children or 15 threads when it started.[1][2][5]
A Secure Pre-threaded and Pre-forked Unix Client-Server Design for Efficient Handling of Multiple Clients

<table>
<thead>
<tr>
<th>Row</th>
<th>Server description</th>
<th>Process control CPU time, (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Iterative server</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>Forking server, one fork per client request</td>
<td>20.90</td>
</tr>
<tr>
<td>2</td>
<td>Pre-fork with each client calling accept.</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td>Pre-fork with parent passing socket descriptor to child</td>
<td>2.58</td>
</tr>
<tr>
<td>4</td>
<td>Threaded server, one thread per client request</td>
<td>0.99</td>
</tr>
<tr>
<td>5</td>
<td>Pre-threaded with mutex locking to protect accept</td>
<td>1.93</td>
</tr>
<tr>
<td>6</td>
<td>Pre-threaded with main thread calling accept</td>
<td>2.05</td>
</tr>
</tbody>
</table>

Table 1: Timing comparisons of the various servers discussed in this paper

9. FORKING SERVER-ONE CHILD PER CLIENT
In this server model we fork parent process or server process into number of server child process and each child process handles one client. The number of clients handles by server is limited on the number of child process for the user ID under which server is running.

The advantage with this model is it can handle multiple clients but amount of CPU time it will takes to fork a child is very high, there is a noticeable delay in launching child processes. Process control CPU time is very high.[1][2][5]

Fig 4: Forking Server

10. PRE-FORKED SERVER- NO-LOCKING AROUND ACCEPT
In this server model we forked the parent process into number of child processes and make a process pool. Each process from available pool handles that client’s request. In this design we regularly monitor processes and maintain number of children to a threshold value, because too many children degrade the server performance. The big advantage of this strategy is this mitigates the startup cost for a process, by creating pool of already forked child processes when the server starts. This design creates a problem of thundering herd.[1][2][5]

Fig 5: Pre-forking Server

11. THUNDERING HERD PROBLEM
This is the problem which arises when server starts and M children are created and all M call accept and all put to sleep. When first connection arrives all M are awakened. Because all M gone to sleep on the same “wait channel”. The first of the M obtain a connection while remaining M-1 will go back to sleep. This situation when all M are awakened even though only one will obtain connection is referred to thundering herd problem, and will degrade the server performance.[1]

12. PRE-FORKED SERVER- DESCRIPTOR PASSING
In this server design only parent or server process calls accept and passed the connected socket to one child process. This resolves the problem of thundering herd by putting lock around the call to accept to all children. This strategy is complicated because this involves passing of descriptor from parent to the children and keeps record of which children are busy and which are free.

Fig 6: Pre-forked server stream pipe
A Secure Pre-threaded and Pre-forked Unix Client-Server Design for Efficient Handling of Multiple Clients

After all children created, listening socket are closed, as only parent calls accept. Parent multiplexed all these descriptors because it handle listening socket along with stream pipes. [1][2][5]

**13. THREADED SERVER-ONE THREAD PER CLIENT**

This is very similar to forking server design. In this threaded server we create a number of server child threads and each thread handle a single client at a time. Thus for a server that can support threads this design is very useful because thread provide a fast context-switch time. [1][2]

**14. PRE-THREADED SERVER – MUTEX LOCKING AROUND ACCEPT**

This server design is very similar to pre-forked design here we make a thread pool instead of process pool when the server starts and then let each thread call accept. Using pre-threading we speedup and enhanced the server performance. We use mutex locking that allow only one thread at a time to call accept. [1][2]

**15. PRE-THREADED SERVER - MAIN THREAD CALLING ACCEPT**

This design is similar to pre-forked server – description passing main thread calls accept and passes the each client connection to one of the available threads in the pool. But there's is no need to pass descriptor from one thread to another since all descriptors are in the same process. All receiving threads needs to know is the descriptor number. [1][2]

**16. HYBRID OF PRE-THREADED AND PRE-FORKED CLIENT SERVER DESIGN**

To achieve speed of pre-threading and stability of pre-forking we combined the two strategies. Both pre-forked server and pre-threaded server design have some advantages and disadvantages. A multiprocess server doesn't crash when one of the child processes crashes, but suffers from slower context switching. Multithreaded servers, on the other hand, do crash when one of the child threads crashes, but have faster context switching. In this hybrid design we pre-forked number of server processes. However these processes do not handle client connections. Each child spawns a finite number of threads, now threads handle client connections, thus while dealing with multiple clients, the server enjoys a fast context switching. If suppose a thread crashes. Only related server child process crashes or affected instead of whole server. We use pre-forked server design with descriptor passing and pre-threaded server design with mutex locking to protect accept. [1][2]

**Fig8: Hybrid of Pre-threaded and Pre-forked server**

**17. COMPARATIVE ANALYSIS OF VARIOUS DESIGN STRATEGIES**

<table>
<thead>
<tr>
<th>Design Strategy</th>
<th>Number of connections</th>
<th>Frequency of New Connections</th>
<th>Stability</th>
<th>Context Switching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forking</td>
<td>Large</td>
<td>Time required to start a new process</td>
<td>Single client cannot crash the server.</td>
<td>Not as fast as thread</td>
</tr>
<tr>
<td>Threading</td>
<td>Large, not as large as forking</td>
<td>Time required to start a new thread.</td>
<td>Single client can crash the server.</td>
<td>Fast</td>
</tr>
<tr>
<td>Pre-forking</td>
<td>Depends on the size of the process pool</td>
<td>Can handle new connections quickly if the process pool is large enough.</td>
<td>Single client cannot crash the server.</td>
<td>Not as fast as threads.</td>
</tr>
<tr>
<td>Pre-threading</td>
<td>Depends on the size of the thread pool</td>
<td>Can handle new connections quickly if the thread pool is large enough.</td>
<td>Single client can crash the server.</td>
<td>Fast</td>
</tr>
<tr>
<td>Combined Pre-threading &amp; Pre-forking</td>
<td>Depends on pool size</td>
<td>Can handle new connections quickly if pool are large enough.</td>
<td>Single client will crash only its parent process, not the whole server.</td>
<td>Fast</td>
</tr>
</tbody>
</table>

*Table 2: Comparative analysis of various design alternatives*
18. SECURE PRE-FORKING
It has been seen in an multitasking server environment, we frequently use concept of pre-forking to improved performance. But if pre-forked processes run as daemon processes, then they pose security and reliability risks. Suppose if an attacker compromises one of the daemon processes, he can use this process to try to infect other processes more effectively because the daemon processes never die. Thus to avoid this vulnerability, we need to follow some security pattern for secure pre-forking such as:

- Pre-forked processes should be created with a limited lifetime.
- New processes are forked in their places after this predefined time span.
- Older processes must be periodically evicted from the resource pool.

![Diagram of resource pool and secured processes](image)

There are various parameters that is to be considered while defining security pattern like number of requests served by a process idle time of processes type of services. This pattern therefore mitigates the trade-off between performance and security.

19. CONCLUSION
We combined the concept of secure pre-forking and pre-threading to gain the speed of pre-threading and the stability of pre-forking. Processes concern with stability, on the other hand threads with fast context-switching time. A multiprocess server suffers from slower context switching, while a multithreaded server has faster context switching. But in a pre-threaded server if one thread crashes it bring down the whole server. Therefore, we combined the two techniques i.e. pre-threading and pre-forking client-server design for efficient handling of multiple clients by minimize the drawbacks and increase the benefits and advantages of both techniques. In this work we make a hybrid design of pre-forking with parent passing socket descriptor to child because process control CPU time is very less as compared to iterative server and resolve the problem of thundering herd also the forked child doesn't directly interact with the clients and pre-threading with mutex locking to protect accept which also resolve the thundering herd problem and has very less process control CPU time and can frequently and efficiently deals with multiple clients. Thus by combining a pre-forking with descriptor passing and pre-threading with mutex locking along with concept of secure pre-forking a pattern for performance and security we get an hybrid client-server design that can efficiently, effectively and securely handles multiple clients at an instant of time and is very useful for high performance servers.

20. FUTURE WORK
We can also merge two or more design alternatives depends on the server type and examine more parameters to provide a more secure pattern for performance and security to increase the scalability of network servers.

21. REFERENCES
[1]. W. Richard Stevens, Bill Fenner, Andrew M. Rudoff –
[2]. *Unix Network Programming – The Sockets Networking*
[3]. *API ,Volume 1; Third Edition; Pearson Education, 2005*
[5]. M Hafiz. Secure Pre-Forking- A Pattern for Performance and Security, University of Illinois at Urbana-Campaign, 2005
[7]. W. Richard Stevens, –*Unix Network Programming –*
[8]. *Interprocess communication,Volume 2; Second Edition;*
[9]. Prentice -Hall of India.