socketservertcl

a Tcl extension for using SCM_RIGHTS

By
Shannon Noe - FlightAware

Presented at the 24th annual Tcl/Tk conference, Houston Texas,
October 2017

Abstract: Horizontal scaling is used to distribute load over many servers. UNIX sockets establish the connection between client and server in a single queue. This queue is accessed by the accept system call. This paper discusses techniques to distribute the connected socket over multiple processes. A Tcl extension, socketservertcl, provides an implementation of the SCM_RIGHTS technique for load distribution.

1. Introduction

The socketservertcl Tcl extension implements SCM_RIGHTS passing of accepted sockets between processes. A server process accepts TCP connections for a pool of workers. Workers are pre-forked processes. Workers receive accepted TCP sockets by registering a callback procedure. This efficiently follows standard Tcl socket patterns used for Tcl socket API calls.

Full source code is available: https://github.com/flightaware/socketservertcl

2. Overview of TCP server code

All servers based on UNIX sockets use the accept system call. The accept system call takes the next client connection from a socket's listen queue. For each new client connection, the server must call accept. The accept call take the next connection from the system's queue. The accept call provides the file descriptor required for the send and recv system calls.

```
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
```

```
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    // setup the address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr,
             sizeof(serv_addr));

    listen(listenfd, SOMAXCONN);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        while (…) {
            recv(connfd, …) // or read
            send(connfd, …) // or write
        }

        close(connfd);
    }
```

Figure 1 A typical TCP server with sockets

3. TCP accept calls are serial

The accept call on the bound socket is serial. Only one process or thread in a process can accept the next connection. There are many techniques to avoid this limitation. Each of these techniques has its relative strengths and weaknesses.

- Proxy client server traffic to servers with additional sockets.
- Using threads to accept().
- Using processes and system mutexes to coordinate accept() calls.
- Using SO_REUSEPORT and accepting connections in multiple processes.
- Passing socket descriptors between processes with SCM_RIGHTS.

3.1 Proxy client server traffic

A proxy will create a connection to a backend servers for each client to the server. The proxy will copy the messages from the client socket to the backend worker socket. The proxy server will copy backend worker messages to the client socket. A proxy server is simple to implement. The overhead of a single threaded proxy server is the main limitation of this technique. Every message incurs additional latency in the proxy transfer.

3.2 Threaded servers

Threaded servers can accept connections and process the client messages in parallel. The coordination of the accept call can be performed in-processes. This is a very performant way to handle parallel connections. Threaded server cannot provide isolation of client requests. If a single thread corrupts the memory, then it is likely the server will fail and affect multiple clients.

3.3 Sharing a socket with system mutexes.

Forked UNIX processes can share bound sockets. Therefore, multiple processes can all perform accept calls on the same socket. Only one processes will successfully accept the connection. Therefore, it is common to coordinate the accept calls between processes with system mutexes. This is a very efficient pattern for distributing sockets between processes in preferred servers.

Apache HTTPD server uses a system mutex to serialize accept calls. All pre-forked workers have the listening socket. Using the mutex only one server issues accept. Using the same technique as Apache HTTPD would require integration of a system mutex with the select.

3.4. Using SO_REUSEPORT option on the socket

Unix allows the sharing of the port number of the TCP socket. Port sharing is enabled by socket option SO_REUSEPORT. A SO_REUSEPORT socket will have the connections distributed to all processes in accept by the kernel. The kernel handles the distribution of accepted connections over the processes. On Linux the distribution of the connect load depends on IP address. It was challenging to create widely distributed connections with a limited number of IP addresses due to the kernel's address hashing algorithm.

3.5 Socket passing with SCM_RIGHTS

The socket can pass from one process to another. This feature is implemented by the SCM_RIGHTS options. SCM_RIGHTS is an option on sendmsg and recvmsg calls. This flag is specifically implemented for socket passing between processes. The socket must pass between the processes using a UNIX pipe. The server accepts a connection, then uses the sendmsg system call to queue the socket file descriptor into the pipe. Then, one client waiting to read the pipe will receive the socket from a recvmsg system call. All of the worker waiting on the pipe will wake up from select, but only one will read each sent file descriptor.

4 Implementing SCM_RIGHTS socket passing in Tcl

4.1 Syntax of the socketserver command implemented in Tcl

The socketserver command is designed to be similar to the pattern of regular socket cmds. The server socket will have a callback for the accepted connection. Two calls are required one to establish the listening and accept call, another to receive a connection by a callback handler. Each accepted connection requires a call to ::socketserver::socket client to receive another connection. This is the same pattern as using Tcl after events for timers.

```
::socketserver::socket server 8888
::socketserver::socket client -port 8888 handle_accept
```

Figure 2 socketserver API calls

4.2 Tcl implementation details

The client side Tcl code is based on the Tcl C API for file channel events. The client code extracts the file descriptor from the message. Then, the event handling code is familiar C Tcl API code for file descriptor handling.

The server Tcl code uses a daemon background thread. This could be replaced by Tcl file events. The background POSIX thread is very efficient at moving the accepted socket connection into the pipe.

5. Caveats and considerations for use

If you need to control when to listen or accept, then do not use socketservertcl. The socketservertcl accepts connections regardless of the workers' state. When there are more active client than workers, then clients will wait and have a TCP connection established to the server. The clients will be connected and the file descriptors queued in the pipe between the server and workers. In the future this behavior may be enhanced back pressure to the server to control if a connection should accept.

6. Future enhancements

The extension could provide more implementation of handling the SCM_RIGHTS messages in Tcl. This would allow Tcl to control the logic of connection handling.

The background thread can be optionally implemented as Tcl events. Currently the extension requires thread support in the Tcl build.

Windows might provide an equivalent implementation for SCM_RIGHTS.