EECS 398 System Design of a Search Engine

Winter 2021

Lecture 16: LinuxTinyServer

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

details

1. Grading underway on the midterm.
2. Working on publishing the last two homeworks:
   HW 9 Expression parser
   HW 10 LinuxTinyServer
3. Staff will be opening up optional meeting slots with individual teams next week, similar to the meetings you had with me.
4. Today is the end of any lecture content needed to build your engines. This is deliberate. You know everything you need to know and you now have 5 weeks to finish building your engines.

details

4. From here, the content won't be critical to your project and I'm not settled on the topics but they may include duplicate detection shingles, what's beyond basic search, various system design problems, ethics, negotiations (I have some exercises I could let you do in breakout rooms), perhaps a course debrief. I may also cancel lectures to do more one-hour meetings with the teams.

5. The last two lecture slots Mon Apr 19 and Wed Apr 21 will be group presentations. We have 15 teams, so you can't be expected to watch all; I'm expecting we'll have to assign you to breakouts so you only watch 3 or 4 each day.

details

Where you probably are now:

Finishing the crawler

1. You should be close to having a working frontier and a way of deciding what to crawl next.
2. You should be parsing and obeying robots.txt.

Making progress on the index

1. You've probably decided what information you're collecting and how you'll encode it in your index.
2. You should be finishing HashTable and Top10 hw, and starting HashBlob and HashFile.
3. Beginning work on ISRs and the constraint solver

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

A simple web server

You need a web server front end for your engine.

Two parts:

1. The HTTP server.
2. A plugin that can exchange JSON with a webpage.

LinuxTinyServer

A very minimal web server for Linux.

1. It begins listing on a socket for connection requests from browser.
2. Each time it gets a request, it creates a thread with a new socket to talk to the client.
3. If it's a "magic path", it can call a plugin module.
4. Otherwise, it handles GET requests by serving up the specified file if it exists.

LinuxTinyServer takes a port number and a root directory for a website.

```
tcsh-5% head LinuxTinyServer.cpp
// Linux tiny HTTP server.
// Nicole Hamilton  nham@umich.edu

// This variation of LinuxTinyServer supports a simple plugin
interface
// to allow "magic paths" to be intercepted.

// Usage:  LinuxTinyServer port rootdirectory

// Compile with g++ -pthread LinuxTinyServer.cpp -o LinuxTinyServer
// To run under WSL (Windows Subsystem for Linux), must elevate with
tcsh-6% ./LinuxTinyServer
Usage:  ./LinuxTinyServer port rootdirectory
tcsh-7%
```

LinuxTinyServer opens a socket and begins listening for connections.

```
tcsh-6% ./LinuxTinyServer
Usage:  ./LinuxTinyServer port rootdirectory
tcsh-7% ls website
Images  Styles  index.htm
tcsh-8% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000
```

LinuxTinyServer responds with the requested page.

```
tcsh-8% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000

Connection accepted from 127.0.0.1:54690

GET /index.htm HTTP/1.1
Host: localhost
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close

Requested path = /index.htm
Actual path = website/index.htm

HTTP/1.1 200 OK
Content-Length: 8964
Connection: close
Content-Type: text/html
```

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

bind( ), listen( ) and accept( )

The basic steps to a web server:

1. Creates two socket variables, one for listening, the other when a new client connects.
2. Build a sockaddr_in structure specifying internet protocol, port number, any IP address, TCP stream.
3. Binds the socket to that address.
4. Enters a loop where it begins listening.
5. Each time it gets a connection request, it spawns a thread to talk to the client.

bind( )

bind( ) is used to connect a socket to a particular address, protocol and port where it can listen.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

bind( )

bind( ) is used to connect a socket to a particular address, protocol and port where it can listen.

All of that is specified in an addrinfo structure.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);

struct addrinfo {
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   socklen_t        ai_addrlen;
   struct sockaddr *ai_addr;
   char            *ai_canonname;
   struct addrinfo *ai_next;
};
```

listen( )

listen( ) marks the socket as one to be used for accepting incoming connection requests.

The backlog is the maximum queue length of pending connections.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

listen( )

listen( ) marks the socket as one to be used for accepting incoming connection requests.

The backlog is the maximum queue length of pending connections.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

```
SOMAXCONN is a system-configured default
maximum socket queue length.

(Under WSL Ubuntu, it's defined as 128 in
/usr/include/x86_64-linux-gnu/bits/socket.h.)
```

listen( )

Any client anywhere on the web that has your IP and port address can try to connect to you.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

SOMAXCONN is a system-configured default maximum socket queue length.

(Under WSL Ubuntu, it's defined as 128 in /usr/include/x86_64-linux-gnu/bits/socket.h.)

accept( )

accept( ) waits until a client tries to do a connect( ) and then returns a *new* socket file descriptor for talking to the client.

The sockaddr that's returns tells you the client's IP address.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr,
        socklen_t *addrlen);
```

```
int main( int argc, char **argv )
   {
   // Check usage and arguments.

   // Create two sockets, one for listening for new
   // connection requests, the other for talking to each
   // new client.

   // Create socket address structures to go with each
   // socket, filling in details of where we'll listen.

   // Create the listenSocket, specifying that we'll r/w
   // it as a stream of bytes using TCP/IP.

   // Bind the listen socket to the IP address and protocol
   // where we'd like to listen for connections.

   // Begin listening for clients to connect to us.

   // Accept each new connection and create a thread to talk with
   // the client over the new talk socket that's created by Linux
   // when we accept the connection.

   // Close the listen socket.
   }
```

Passing the talk socket to the child

When creating a child thread, you get to pass a void *, usually a pointer to an object with whatever information the child needs.

Since the server expects to get lots of connection requests, it can't pass a pointer to a local or global variable that will quickly be overwritten.

Solution is to pass a pointer to an object on the heap and let the child delete it.

```
while ( ( talkAddressLength = sizeof( talkAddress ),
      talkSocket = accept( ... ) ) && talkSocket != -1 )
   {
   pthread_t child;
   pthread_create( &child, nullptr, Talk,
         new int( talkSocket ) );
   pthread_detach( child );
   }
```

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

The talk thread

The Talk thread looks for a GET message and replies with the requested file.

But it also has a plugin interface that allows a server application to intercept “magic paths”.

```
void *Talk( void *talkSocket )
   {
   // Cast from void * to int * to recover the talk
   // socket id then delete the copy passed on the heap.

   // Allocate a buffer for reading the incoming
   // request  and for reading the requested file.

   // Do a recv( ) to get the request.

   // Parse the request to find the action and path
   // being requested.

   // Watch for a plugin that intercepts this path.

   // If it's a GET and the path is found in the website
   // directory, return it with an HTTP/1.1 200 OK
   // message, otherwise with a 403 or 404.

   // Close the talk socket.
   }
```

The talk thread

It can't just paste the requested path onto the end of the website directory path.

Must watch for ".." segments and forbid access outside the website.

```
void *Talk( void *talkSocket )
   {
   // Cast from void * to int * to recover the talk
   // socket id then delete the copy passed on the heap.

   // Allocate a buffer for reading the incoming
   // request  and for reading the requested file.

   // Do a recv( ) to get the request.

   // Parse the request to find the action and path
   // being requested.

   // Watch for a plugin that intercepts this path.

   // If it's a GET and the path is found in the website
   // directory, return it with an HTTP/1.1 200 OK
   // message, otherwise with a 403 or 404.

   // Close the talk socket.
   }
```

Agenda

1. Course details
2. TinyLinuxServer
3. bind( ), listen( ) and accept( )
4. The Talk( ) thread
5. A plugin interface

The plugin interface

The Talk thread looks for a GET message and replies with the requested file.

But it also has a plugin interface that allows a server application to intercept “magic paths”.

```
class PluginObject
   {
   public:
      // MagicPath returns true if this is a path
      // the plugin intercepts.

      virtual bool MagicPath( string path ) = 0;

      // The request passed to ProcessRequest is
      // the raw contents of the HTTP request as
      // read from the talk socket.

      // Whatever is returned is  written unchanged
      // to the socket (and to the client) with a
      // proper HTTP header.

      string ProcessRequest( string request ) = 0;

      virtual ~PluginObject( )
         {
         }
};
```

The plugin interface

The plugin registers itself by setting a global pointer.

```
// The constructor for any plugin should set
// Plugin = this so that LinuxTinyServer knows
// it exists and can call it.

extern PluginObject *Plugin;
```

The plugin interface

The plugin registers itself by setting a global pointer.

The initial value is null.

```
// The constructor for any plugin should set
// Plugin = this so that LinuxTinyServer knows
// it exists and can call it.

#include "Plugin.h"
PluginObject *Plugin = nullptr;
```

The plugin interface

Example: The new EECS 280 P4 Web project.

The plugin constructor registers itself by setting the global pointer.

https://eecs280staff.github.io/p4-web/

```
class P4_Web : public PluginObject
   {
   public:
      bool MagicPath( const string path )
         {
         // Return true if this is a path that
         // this plugin intercepts.
         }
      string ProcessRequest( const string request )
         {
         // Read the request and return a string
         // with the proper HTTP header and content.
         }
      P4_Web( )
         {
         // Register this plugin.
         Plugin = this;
         }
      ~P4_Web( )
         {
         }
};
```