Оглавление

Глава 14

Программирование с сокетами

Наиболее интересные программы, которые я пишу так или иначе включают сокеты. Сокет – это конечная точка соединения, которая позволяет взаимодействовать машинам по Интернет, используя Internet Protocol(IP). В этом разделе мы сконцентрируем свое внимание на двух протоколах интернета: Transmission Control Protocol(TCP) и User Datagram Protocol(UDP)

UDP позволяет приложениям посылать друг другу короткие сообщения(называемые дейтаграммами), но этот протокол не гарантирует доставку сообщений. Дейтаграммы могут прийти в неправильном порядке. С дргой стороны – TCP, предоставляет надежный поток байтов, которые доставляются в правильном порядке на протяженни всего соединения.

Почему программирование с использованием сокетов увлекательно? Потому что это позволяет приложениям взаимодействовать по интернет с другими машинами, что имеет гораздо больший потенциал, чем выполнение локальных операций.

Существуют две основные библиотеки для программирования на сокетах – это get_tcp, для программирования TCP соединений, и gen_udp для UDP соединений

В этой главе мы увидим как клиент и сервер используют TCP и UDP сокеты. Мы пройдем через различные формы серверов: параллельные, последовательные, блокирующий и неблокирующие, и увидим, как сделать traffic-shaping приложения, которые контроллируют поток данных к программам.

14.1 Использование TCP

Мы начнем наше путешествие в программирование сокетов с рассмотрения простой TCP программы, которая получает данные с сервера. После этого мы напишем простой последовательный TCP сервер, и покажем, как он может быть распараллелен для обработки множества параллельных сессий.

14.1.1 Получение данных с сервера

Начнем с написания небольшой функции[1] [2], которая использует TCP сокет для получения HTML страницы с http://www.google.com:
Download socket_examples.erl

```
nano_get_url() ->
    nano_get_url("www.google.com").

nano_get_url(Host) ->
    {ok,Socket} = gen_tcp:connect(Host,80,[binary, {packet, 0}]),
    ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
    receive_data(Socket, []).
```

[1] стандартная библиотечная функция, которая делает то же самое называется http:request(Url). Но мы хотим показать, как это можно сделать средствами библиотеки gen_tcp.

[2] В современной версии документации нету библиотеки http, зато есть httpc

```
receive_data(Socket, SoFar) ->
    receive
{tcp,Socket,Bin} ->
    receive_data(Socket, [Bin|SoFar]);
{tcp_closed,Socket} ->
    list_to_binary(reverse(SoFar))
    end.
```

Как это работает?

1. Мы открываем TCP сокет с адресом http://google.com на 80 порту, при помощи gen_tcp:connect. Аргумент binary в функции connect говорит системе открыть сокет в двоичном режиме и доставлять все данные приложению как бинарные. {packet,0} в контексте TCP означает, что данные доставляются непосредственно приложению, в немодифицированной форме.

2. Мы вызываем get_tcp:send и посылаем сообщение GET / HTTP/1.0\r\n\r\n в сокет. Затем мы ожидаем ответа. Ответ не придет весь одним пакетом, он прийдет фрагментами. Процесс, открывший сокет, или контроллирующий его будет получать фрагменты, как последовательность сообщений.

3. Мы принимаем сообщение вида {tcp,Socket,Bin}. Третий аргумент этого кортежа – это двоичные данные.Так получилось потому, что мы открыли сокет в бинарном режиме. Это сообщение — один из фрагментов, которые веб-сервер посылает нам. Мы получили один фрагмент, добавили его в список фрагментов, и затем ожидаем следующий фрагмент.

4. Мы получаем {tcp_closed,Socket}. Это произошло потому, что сервер закончил отправку данных. [3]

5. Когда все фрагменты пришли,нам необходит реверсировать список, поскольку мы сохраняли фрагменты в неправильном порядке.

Давайте проверим, что это работает:

```
1> B = socket_examples:nano_get_url().
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"...>>
```

Примечание: Когда вы запускаете *nano_get_url* , то результат будет двоичный. Таким образом вы увидите, что двоичные данные выглядят, как при "pretty printed"в эрланговской оболочке. Когда двоичные данные печатаются в формате "pretty printed все управляющие символы выводятся в escape-формате.Банарные данные выводятся не полностью, это видо по трем точкам (...>>) в конце печати. Если вы желаете увидеть все бинарные данные, можно использовать *io:format* , или разорвать банарные данные на символы, при помощи *string:tokens* :

```
2> io:format("~p~n",[B]).
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
   Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"
   TM=176575171639526:LM=1175441639526:S=gkfTrK6AFkybT3;
   expires=Sun, 17-Jan-2038 19:14:07
   ... several lines omitted ...
>>

3>string:tokens(binary_to_list(B),"\r\n").
["HTTP/1.0 302 Found",
```

[3] Это верно только для HTTP/1.0; для более новых версий используются другие стратегии

```
"Location: http://www.google.se/",
"Cache-Control: private",
"Set-Cookie: PREF=ID=ec7f0c7234b852dece4:TM=11713424639526:
LM=1171234639526:S=gsdertTrK6AEybT3;
expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com",
"Content-Type: text/html",
"Server: GWS/2.1",
"Content-Length: 218",
"Date: Fri, 16 Feb 2007 15:25:26 GMT",
"Connection: Keep-Alive",
... lines omitted ...
```

Это более или менее показывает, как работает web-клиент(с уклонением в меньшую сторону — приходится делать много работы, для корректного выведения результата в веб-браузерах). Предыдущий код, тем не менее, — хорошая отправная точка для ваших собственных экспериментов. Вам может понравиться модифицировать этот код, например: вы можете принимать и хранить записи веб-сайтов или автоматически прочитать ваш ваш e-mail. Возможности безграничны.

Заметим, что код, который собирает фрагменты выглядел так:

```
receive_data(Socket, SoFar) ->
    receive
{tcp,Socket,Bin} ->
    receive_data(Socket, [Bin|SoFar]);
{tcp_closed,Socket} ->
    list_to_binary(reverse(SoFar))
    end.
```

Таким образом мы добавляем прибывшие фрагменты в голову списка *SoFar* . Когда все фрагменты прибыли и сокет был закрыт мы реверсируем список и соединяем фрагменты.

Вы могли подумать, что сборка фрагментов таким образом было бы лучшим решением:

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
    receive_data(Socket, list_to_binary([SoFar,Bin]));
{tcp_closed,Socket} ->
    SoFar
    end.
```

Этот код корректен, но менее эффективен, чем оригинальная версия. Причина этого состоит в том, что этот код беспрестанно добавляет новые данные в конец буфера, а это включает большое копирование данных. Куда лучше добавлять фрагменты в голову списка, а потом реверсировать записи списка и собрать все фрагменты одной операцией.

14.1.2 Простой TCP сервер

В предыдущем разделе мы написали простой клиент. Давайте теперь напишем сервер.

Сервер открывает порт 2345 и ждет одного сообщения. Это двоичное сообщение, которое содержит терм эрланга. Терм – это эрданговская строка, содержащая выражение. Сервер обрабатывает выражение и посылает результат клиенту, записывая результат в сокет.

Как мы можем писать веб сервер?

Написание что-то вроде веб-клиента или сервера очень интересно. Действительно, многие люди уже имеют написанные эти вещи, но если вы действительно хотите понять,как это работает, копайте глубже и выяснейте. Это очень поучительно. Кто знает — может быть наш веб-сервер будет самый лучший. Так как нам начать?

Для создания веб-сервера, или другого программного обеспечения, которое реализует один из стандартных протокол интернета, вам необходимы правильные инструменты, а так же необходимо четко понимать, как точно этот протокол реализовать.

В нашем примере есть код, который получает веб-страницу. Откуда мы узнали, что надо открывать 80ый порт. Откуда мы узнали, что серверу надо посылать именно такое сообщение: GET / HTTP/1.0\r\n\r\n? Ответ прост. Все основные протоколы для интернет сервисов описаны в request for comments(RFCs). HTTP/1.0 описан в RFC 1945. Официальный веб-сайт для всех документов RFC — http://www.ietf.org(дом для Internet Engineering Task Force).

Другой бесценный источник информации - сниффер. При помощи сниффера мы можем захватывать и анализировать все IP пакеты приходящие и уходящие от приложения. Большое количество сниферов включают программное обеспечение, которое может декодировать и анализировать данные в пакете, а так же представлять данные в выразительной форме. Один из наиболее известных и возможно лучших снифферов — это Wireshark(Ранее известный, как ethereal), доступен на http://www.wireshark.org.

Вооружившись пакетным сниффером и соответствующими документами RFC, мы готовы писать следующие убийственные приложения.

Написав эту программу(), мы сможем ответить на несколько простых вопросов:
1)Как организованы эти данные?Как мы узнаем, сколько данных составляет один запрос или ответ?
2)Как эти данные кодируются и декодируются в пределах запроса или ответа?

Данные TCP сокета — это просто недифференцируемый поток байтов. В процессе доставки эти данные могут быть фрагментированы. Поэтому нам нужно некоторое соглашение, что бы мы знали сколько байтов составляет единичный запрос или ответ.

В случае Эрланга мы используем простое соглашение, по которому перед каждым запросом или ответом мы приписываем перед ним 1,2 или 4 байта, которые характеризуют его длину. Это количество байт передается функциям get_tcp:connect и gen_tcp:listen,как аргумент {packet,N} [4]. Заметим, что аргумент packet должен быть согласован между клиентом и сервером. Если сервер откроет соединение с {packet,2}, а клиент с {packet,4}, то ничего работать не будет.

Имея открытый сокет с опцией {packet,N}, мы не должны беспокоиться о фрагментации данных. Драйвер эрланга ,до передачи сообщения нашей программе, удостоверится, что все фрагменты данных собраны с правильной длиной.

Следующий интерес представляют кодирование и декодирование данных. Мы будем использовать простейший возмодный путь кодирования и декодирования сообщений, используя *term_to_binary* для кодирования и *binary_to_term* для декодирования.

Заметим, что соглашение упаковки({packet,N}) и правила кодирования, необходимые для общения клиента и сервера, достигаются в двух строках кода: используя {packet,4}, когда мы открываем сокет, и *term_to_binary* — для кодирования.

Легкость, с которой мы можем упаковывать и кодировать эрланговские термы, дает нам значительное приемущество над text-based методами, такими как HTTP или XML. Используя эрланговский BIF *term_to_binary* и его обратный *binary_to_term* ,обычно, на порядок быстрее, чем вычисление эквивалентных операций, которые использует XML термы и включают пересылку намного большего количества данных. А теперь к программам. Во-первых, вот очень простой сервер.
Download socket_examples.erl

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},  %% (6)
 {reuseaddr, true},
 {active, true}]),
    {ok, Socket} = gen_tcp:accept(Listen),  %% (7)
    gen_tcp:close(Listen),  %% (8)
    loop(Socket).

loop(Socket) ->
```

[4] Директива *packet* здесь означает не сколько физически байтов будет записываться в сокет, а именно длину сообщения в программе

```
    receive
{tcp, Socket, Bin} ->
    io:format("Server received binary = ~p~n",[Bin]),
    Str = binary_to_term(Bin),  %% (9)
    io:format("Server (unpacked)  ~p~n",[Str]),
    Reply = lib_misc:string2value(Str),  %% (10)
    io:format("Server replying = ~p~n",[Reply]),
    gen_tcp:send(Socket, term_to_binary(Reply)),  %% (11)
    loop(Socket);
{tcp_closed, Socket} ->
    io:format("Server socket closed~n")
    end.
```

Как это работает?

1. Вначале мы вызываем *gen_ tcp:listen* , для прослушивания 2345 порта, и устанавливаем соглашение об упаковке. {packet,4} подразумевает, что сообщению будет предшествовать 4х байтовый заголовок.

   Затем *gen_ tcp:listen(...)* возвращает {ok,Socket} или {error,Why}, но нас интересует только тот случай, когда мы можем открыть сокет. Поэтому мы пишем следующий код:

   ```
   {ok, Listen} = gen_tcp:listen(...),
   ```

   Здесь происходит сопоставление по шаблону, и если *get_ tcp:listen* возвратит {error, ...}, то будет поднято исплючение. В случае успеха, это выражение связывает *Listen* с прослушиваемым сокетом, и он используется в качестве аргумента в *geb_ tcp:accept*

2. Теперь мы вызываем *gen_ tcp:accept(Listen)* . В этом месте программа усыпляется и ожидает соединения. Когда соединение установлено, эта функция возвращает переменную Socket, связанную с сокетом, который может использоваться для общения с клиентом, который установил соединение.

3. Когда функция *accept* возвращает управление, мы сразу же вызываем *gen_ tcp:close(Listen)*. Функция *close* закрывает прослушиваемый сокет, после чего сервер становится недоступным для других соединений. Это не окахывает эффекта на имеющееся соединение; это только предотвращает новые соединения.

4. Мы декодируем входные данные.

5. Затем мы вычисляем строку.

6. И наконец, мы кодируем ответ и посылаем его обратно в сокет.

Заметим, что эта программа принимает только один единственный запрос, потом она завершится и больше не будет принимать соединений.

Это простейший пример сервера, иллиюстрирующий то, как упаковываются и коидруются данные. Этот код принимает запрос, вычисляет ответ, отправляет ответ, и завершается.

Для тестирования сервера нам потребуется соответствующий клиент:
Download socket_examples.erl

```
nano_client_eval(Str) ->
    {ok, Socket} =
gen_tcp:connect("localhost", 2345,
[binary, {packet, 4}]),
    ok = gen_tcp:send(Socket, term_to_binary(Str)),
    receive
{tcp,Socket,Bin} ->
    io:format("Client received binary = ~p~n",[Bin]),
    Val = binary_to_term(Bin),
```

```
    io:format("Client result = ~p~n",[Val]),
    gen_tcp:close(Socket)
    end.
```

Для тестирования нашего кода мы запустим клиент и сервер на одной машине, поэтому адрес хоста жестко прописан в функции *gen_ tcp:connect* , как localhost.

Заметим, что *term_ to_ binary* вызывается клиентом для кодирования сообщения и *binary_ to_ term* вызывается сервером, для переконструирования пришедшего сообщения.

Для запуска этого кода нам потребуется открыть два терминала и запустить эрланговскую оболочку в каждом их них.

В начале мы запустим сервер:

```
1> socket_examples:start_nano_server().
```

Мы не увидим другого вывода, пока ничего не происходит. Затем перейдем к клиенту и напишем следующую команду:

```
1> socket_examples:nano_client_eval("list_to_tuple([2+3*4,10+20])"
```

В окне с сервером мы должны увидеть следующее:

```
Server received binary = <<131,107,0,28,108,105,115,116,95,116,
   111,95,116,117,112,108,101,40,91,50,
   43,51,42,52,44,49,48,43,50,48,93,41>>
Server (unpacked) "list_to_tuple([2+3*4,10+20])"
Server replying = {14,30}
```

В окне с клиентом мы должны увидеть такой текст:

```
Client received binary = <<131,104,2,97,14,97,30>>
Client result = {14,30}
ok
```

И в конце концов в серверном окне будет вот так:

```
Server socket closed
```

14.1.3 Улучшение сервера

В предыдущем разделе мы сделали сервер, который принимает только одно соединение,после чего завершается. Чуть-чуть изменив код мы можем получить два различных вида серверов:

1. Последовательный — принимает одное соединение единовременно.
2. Параллельный сервер — множество параллельных соединений единовременно.

Изначальная версия кода выглядит так:

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket).
    ...
```

Будем изменять этот код, и получим два варианта серверов.

14.1.4 Последовательный сервер

Для создания последовательного сервера мы изменим код следующим образом:

```
start_seq_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    seq_loop(Listen).
seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket),
    seq_loop(Listen).
    loop(..) -> %% по-старому
```

Этот код работает почти как и предыдущий, но так как мы хотим обрабатывать больше одного запроса, мы оставляем прослушиваемый сокет открытым и не вызываем *gen_ tcp:close(Listen)* . Другое отличие, что после того, как loop(Socket) завершится, мы вызываем seq_loop(Listen) снова, где и ожидается следующее соединение.

Если клиент попытается соединиться с сервером, пока он занят с имеющимся соединением, то тогда он будет поставлен в очередь, пока сервер не закончит обработку текущего соединения. Если число соединений в очереди будет превышать значение listen backlog, тогда соединения будут отвергаться.

Мы показали код, который только запускает сервер. Останов сервера прост(как и останов параллельного); просто убейте процесс, который запускал сервер или серверы. gen_tcp связывает себя с контроллируемыми процессами. И если контроллируемый процесс умирает, то это закрывает сокет.

14.1.5 Параллельный сервер

Трюк создания параллельного сервера немедленно порождает дочерний процесс, когда gen_tcp:accept получает новое соединение:

```
start_parallel_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    spawn(fun() -> par_connect(Listen) end).
par_connect(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen) end),
    loop(Socket).
loop(..) -> %% как и раньше
```

Этот код схож с последовательным сервером, который вы видели ранее. Решающее различие заключается в добавлении функции spawn, которая гарантирует создание параллельных процессов, для каждого нового соединения. Вы должны взглянуть на место, где стоит *spawn* и увидеть, как эта функция превращает последовательный сервер в параллельный.

Все эти сервера вызывают gen_tcp:listen и gen_tcp:accept; единственное различие заключается в том, что мы называем эти функции параллельной программой или последовательной программой.

14.1.6 Заметки

Будем осведомлены о следующем:

- Процесс, который создает сокет (вызывая gen_tcp:accept или gen_tcp:connect) называется процессом, контроллирующем этот сокет. Все сообщения из сокета будут отправляться контроллирующему процессу;Если контроллирующий процесс умирает, тогда сокет будет закрыт. Контроллирующий процесс может быть изменен на NewPid при помощи вызова gen_tcp:controlling_process(Socket,NewPid).

- Наш сервер потенциально может установить многие тысячи соединений. Возможно, мы захотим ограничить максимальное число одновременных соединений. Это может быть реализовано при помощи счетчика того, сколько соедиений сейчас установлено. Мы инкрементируем его, если поступает новое соединение, и декрементируем, если соединение завершается. Мы можем использовать этот механизм для ограничения общего числа одновременных соединений в системе.

- После принятия соединения хорошей идеей будет явное задание необходимых опций сокета,вот так:

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet,4},binary,
      {nodelay,true},{active, true}]),
loop(Socket)
```

- В версии эрланга R11B-3 раличным процессам позволено вызывать gen_tcp:accept для одного и того же сокета. Это простейший пример параллельного сервера, поскольку мы можем иметь кучу заранее порожденных процессов, каждый из которых будет ожидать соединения при помощи gen_tcp:accept/1.

14.2 Контроллирование проблемы

Эрланг-сокет может быть открыт в одном из трех режимов: активный, единыжды активный, пассивный. Это достигается включением опции {active,true|false|once} в аргумент Options одной из двух функций gen_tcp:connect(Adress,Port,Options) или gen_tcp:listen(Port,Options).

Если указано {active,true}, тогда будет создан активный сокет;{active,false} указывает на создание пассивного сокета. {active,once} создает сокет, который будет активным, но только до приема одного сообщения; после того, как сообщение будет принято, сокет сделается пассивным до того момента, как сможет принять новое сообщение.

В следующих разделах мы посмотрим, как применяются эти различные виды сокетов.

Различие между активным и пассивным режимом заключается в том, как происходит прием сообщения сокетом.

- Если был создан активный сокет, тогда контроллирующему процессу будут приходить кортежи вида {tcp,Socket,Data} в почтовый ящик. В этом случае контроллирующий процесс никак не сможет контроллировать поток сообщений. Злоумышленник может отправить тысячи сообщений системе, и все они будут доставлены контроллирующему процессу. Контроллирующий процесс никак не сможет сотановить этот поток сообщений.

- Если сокет был открыт в пассивном режиме, тогда для приема сообщений с сокета контроллирующий процесс вызывает gen_tcp:recv(Socket,N). Этот вызов будет пытаться получить ровно N байт из сокета. Если N = 0, тогда все доступные байты будут возвращены.В этом случае сервер может контроллировать поток байтов от клиента, выбирая, когда использовать gen_tcp:recv.

Пассивный режим используется для контроллирования потока, который идет на сервер. Для иллюстрации этого мы можем написать функцию по приему сообщений в трех видах:

- Активный прием сообщений(неблокирующий)

- Пассивый прием сообщений(блокирующий)

- Гибридный прием сообщений(частичное блокирование)

14.2.1 Активный прием сообщений(неблокирующий)

В нашем первом примере мы открываем сокет в активном режиме и затем принимаем сообщения с сокета:

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, true}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
{tcp, Socket, Data} ->
    ... делаем что-то с Data ...
{tcp_closed, Socket} ->
    ...
    end.
```

Этот процесс не может контроллировать поток сообщений к серверу. Если клиент производит данные быстрее, чем сервер может обработать, тогда система может быть зафлужена(flooded) сообщениями — буфер сообщений заполнится и система может упасть или вести себя странно.

Этот тип сервера называется неблокирующим сервером, потому что он не может блокировать клиента. Мы должны писать неблокирующие сервера, только в том случае, когда мы можем быть уверены, что сервер сможет справиться с запросами клиентов.

14.2.2 Пассивный прием сообщений(блокирующий)

В этом разделе мы напишем блокирующий сервер. Сервер открывает сокет в пассивном режиме, устанавливая опцию {active,false}. Сервер не может обрушиться из-за гиперактивного клиента, который пытается зафлудить его большим количеством данных.

Код в функции loop вызывает gen_tcp:recv все время, когда нужно принять данные. Клиент будет заблокирован, пока сервер вызывает recv. Заметим,что ОС тоже буферизует данные, что позволяет клиенту отправить большое количество данных пока он заблокирован, даже если recv не вызывалась.

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, false}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    case gen_tcp:recv(Socket, N) of
{ok, B} ->
    ... делаем что-то с данными ...
    loop(Socket);
{error, closed}
    ...
end.
```

14.2.3 Гибридный подход(частичное блокирование)

Вы можете подумать,что использование пассивного режима для всех серверов — это корректный подход. К сожалению, это не так, когда мы были в пассивном режиме, мы могли ожидать данные только с одного сокета. Это не годится для написания серверов, которые должны ожидать данные со многих сокетов. К счастью, мы можем применить гибридный подход, когда никто ни неблокирующий, ни блокирующий. Мы открываем сокет с опцией {active,once}. В этом режиме сокет активен, но только до первого сообщения. После того,как контроллирующему процессу было послано сообщение, необходимо вызвать inet:setops, что бы включить прием следующего сообщения. Система будет блокировать прием, пока это не произойдет. Это лучше, чем два других метода. Вот как выглядит этот код:

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, once}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
{tcp, Socket, Data} ->
    ... do something with the data ...
    %% когда вы готовы принять следующее сообщение
    inet:setopts(Sock, [{active, once}]),
    loop(Socket);
{tcp_closed, Socket} ->
    ...
end.
```

Используя {active,once} опцию пользователь может осуществлять продвинутые формы контроля потока(иногда это называется traffic-shaping) и ,таким образом, предотвращать зафлужевание(flooded) сервера чрезмерными сообщениями.

14.3 Откуда это соединение к нам пришло?

Допустим, мы написали некоторый вид онлайн сервера и заметили,что кто-то спамит наш сайт. Как мы можем на это отреагировать? Первым делом необходимо узнать, откуда поступило это соединение. Чтобы это определить мы можем использовать inet:peername(Socket).

@spec inet:peername(Socket) -> {ok, {IP_Address, Port}} | {error, Why}
Эта функция возвращает IP адрес и порт другого конца соединения, таким образом сервер может определить кто инициировал соединение. IP_Adress это кортеж из целых чисел,для IPv4 имеет вид {N1,N2,N3,N4}, а для IPv6 {K1,K2,K3,K4,K5,K6}. N_i и K_i числа в дипазоне от 0 до 255.

14.4 Обработка ошибок сокетов

Обработка ошибок сокета это очень просто, по существу вам ничего не надо делать. Как говорилось ранее, каждый сокет имеет контроллирующий процесс(тоесть процесс, который создал сокет). Если контроллирующий процесс умирает, тогда сокет автоматически хакрывается.

Это значит,что если, например, вы имеете клиент и сервер, и сервер падает из-за программной ошибки, то сокет, который принадлежил серверу будет автоматически закрыт, и клиенту будет послан кортеж {tcp_closed,Socket}.

Мы можем протестировать этот механизм со следующий программой:
Download socket_examples.erl

```
error_test() ->
    spawn(fun() -> error_test_server() end),
    lib_misc:sleep(2000),
    {ok,Socket} = gen_tcp:connect("localhost",4321,[binary, {packet, 2}]),
    io:format("connected to:~p~n",[Socket]),
    gen_tcp:send(Socket, <<"123">>),
    receive
Any ->
    io:format("Any=~p~n",[Any])
    end.

error_test_server() ->
    {ok, Listen} = gen_tcp:listen(4321, [binary,{packet,2}]),
```

```
    {ok, Socket} = gen_tcp:accept(Listen),
    error_test_server_loop(Socket).

error_test_server_loop(Socket) ->
    receive
{tcp, Socket, Data} ->
    io:format("received:~p~n",[Data]),
    atom_to_list(Data),
    error_test_server_loop(Socket)
    end.
```

Когда мы запустим ее, мы увидим следующее:

```
1> socket_examples:error_test().
connected to:#Port<0.152>
received:<<"123">>
=ERROR REPORT==== 9-Feb-2007::15:18:15 ===
Error in process <0.77.0> with exit value:
  {badarg,[{erlang,atom_to_list,[<<3 bytes>>]},
  {socket_examples,error_test_server_loop,1}]}
Any={tcp_closed,#Port<0.152>}
ok
```

при помощи spawn мы породили сервер, затем усыпились на 2 секунды(что бы сервер успел запуститься), и затем отправляем сообщение, содержащее `<<"123">>`. Когда это сообщение приходит, сервер пытается вычислить atom_to_list(Data), где Data — это бинарные данные, и немедленно падает[5]. Теперь, когда контроллирующий процесс(со стороны сервера) обрущился, сокет автоматически закрывается. Затем клиенту отправляется сообщение {tcp_closed,Socket}.

14.5 UDP

Теперь давайте рассмотрим User Datagram Protocol (UDP). Используя UDP, машины могут отправлять друг другу коротенькие сообщения по интернет, которые называются дейтаграммы. UDP дейтаграммы ненадежны. Это значит что если клиент отправил последовательность UDP дейтаграм серверу, то они могут прийти не в том порядке,в каком отправлялись, могут прийти не все, или дейтаграммы могут продублироваться, но если одна дейтаграмма пришла на сервер, то она будет неповрежденной. Большие дейтаграммы могут быть разбиты на маленькие фрагменты, но IP протокол будет собирать фрагменты, перед передачей приложению.

UDP устроен так, что до отправки дейтаграммы между клиентом и сервером не установлено соединение. Это значит,что UDP хорошо преспособлен для приложений, у которых большое число клиентов, которые отправляют короткие сообщения серверу.

Создание UDP клиента и сервера в Эрланге еще проще, чем создание клиента или сервера на TCP, если мы не беспокоимся о поддержке соединения.

14.5.1 Простейший UDP сервер и клиент

Давайте вначале обсудим сервер. Основной вид UDP сервера следующий:

```
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    loop(Socket).

loop(Socket) ->
```

[5]системный монитор печатает диагностическое сообщение, которое вы видите в оболочке

```
    receive
{udp, Socket, Host, Port, Bin} ->
    BinReply = ... ,
    gen_udp:send(Socket, Host, Port, BinReply),
    loop(Socket)
    end.
```

Этот код, отчасти, проще кода с TCP, поскольку мы не беспокоимся о приеме сообщения "socket closed". Заметим, что открыли сокет в бинарном режиме, что говорит драйверу отправлять все сообщения контроллирующему процессу, как бинарные данные.

Теперь клиент. Тут просто открывается UDP сокет, отправляется сообщение серверу, ожидается ответ(или таймаут), и затем закрывается сокет и возвращается значение, которое пришло от сервера.

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    ok = gen_udp:send(Socket, "localhost" , 4000, Request),
    Value = receive
{udp, Socket, _, _, Bin} ->
    {ok, Bin}
after 2000 ->
    error
    end,
    gen_udp:close(Socket),
    Value
```

Мы должны выставить таймаут, поскольку UDP ненадежный, мы можем просто не получить ответа.

14.5.2 UDP факториал сервер

Мы легко можем создать модуль UDP сервера, который подсчитывает факториал любого целого числа, которого ему отправили. Код сделан по аналогии с предыдущим разделом.
Download udp_test.erl

```
%% Сервер
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    io:format("server opened socket:~p~n",[Socket]),
    loop(Socket).

loop(Socket) ->
    receive
{udp, Socket, Host, Port, Bin} = Msg ->
    io:format("server received:~p~n",[Msg]),
    N = binary_to_term(Bin),
    Fac = fac(N),
    gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
    loop(Socket)
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).

%% Клиент

client(N) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
```

```
    io:format("client opened socket=~p~n",[Socket]),
    ok = gen_udp:send(Socket, "localhost", 4000,
      term_to_binary(N)),
    Value = receive
{udp, Socket, _, _, Bin} = Msg ->
    io:format("client received:~p~n",[Msg]),
    binary_to_term(Bin)
    after 2000 ->
    0
    end,
    gen_udp:close(Socket),
    Value.
```

Обратите внимание,что я добавил несколько отладочных печатей, что бы вы увидели что происходит, когда запускается программа.Я всегда добавляю несколько отладочных печатей, когда разрабатываю программу, и затем комментирую или редактирую их во время работы программы.

Теперь давайте запустим этот пример. Вначале запустим сервер.

```
1> udp_test:start_server().
server opened socket:#Port<0.106>
<0.34.0>
```

Он запускается в фоновом режиме, теперь мы можем сделать клиентский запрос:

```
2> udp_test:client(40).
client opened socket=#Port<0.105>
server received:{udp,#Port<0.106>,{127,0,0,1},32785,<<131,97,40>>}
client received:{udp,#Port<0.105>,
  {127,0,0,1}, 4000,
  <<131,110,20,0,0,0,0,0,64,37,5,255,
  100,222,15,8,126,242,199,132,27,
  232,234,142>>}
  81591528324789773434561126959611589427200000000
```

14.5.3 Дополнительные замечания про UDP

Мы должны отметить, что UDP не устанавливает соединения между клиентом и сервером, сервер не имеет методов блокировать клиента, отвергая чтение данных —сервер не имеет представления, кто является клиентом.

Большие UDP пакеты могут фрагментироваться, по мере прохождения через сеть. Фрагментация имеет место, когда размер UDP данных превышает величину maximum transfer unit (MTU), эта величина определяется узлами сети, через которые проходит пакет. Обычно, при настройке сети, рекомндуют на начальном этапе выставить MTU около 500 байтов, и затем постепенно увеличивать с измерение пропускной способности сети. Если на некотором узле пропускная способность резко упадет, тогда вы узнаете, что пакет является очень большим.

UDP пакеты могут быть доставлены дважды(что удивит некоторых людей), таким образом вы должны быть осторожны, когда пишете код для удаленных процедурных вызовов. Это может произойти, если ответ сервера на второй запрос совпадает с ответом сервера на первый запрос. Что бы избежать этого мы должны модифицировать клиентский код, включением уникальных идентификаторов, и проверкой, что сервер возвращает этот идентификатор. Для генерации уникальных идентификаторов, мы вызываем BIF make_ref, которая гарантирует возвращение глобального уникального идентификатора. Код для удаленного процедурного вызова теперь выглядит так:

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    Ref = make_ref(), %% создание уникального идентификатора
```

```
    B1 = term_to_binary({Ref, Request}),
    ok = gen_udp:send(Socket, "localhost" , 4000, B1),
    wait_for_ref(Socket, Ref).

wait_for_ref(Socket, Ref) ->
    receive
{udp, Socket, _, _, Bin} ->
    case binary_to_term(Bin) of
{Ref, Val} ->
    %% получено корректное число
    Val;
{_SomeOtherRef, _} ->
    %% пришло какое-то другое число. отбрасываем его.
    wait_for_ref(Socket, Ref)
    end;
after 1000 ->
    ...
    end.
```

14.6 Широковещание на множество машин

Для полноты, я покажу вам, как создавать широковещательный канал. Этот код вряд ли вам пригодится, но возможно, что в один день вам он понадобится.
Download broadcast.erl

```
send(IoList) ->
    case inet:ifget("eth0", [broadaddr]) of
{ok, [{broadaddr, Ip}]} ->
    {ok, S} =  gen_udp:open(5010, [{broadcast, true}]),
    gen_udp:send(S, Ip, 6000, IoList),
    gen_udp:close(S);
_ ->
    io:format("Bad interface name, or\n"
      "broadcasting not supported\n")
    end.

listen() ->
    {ok, _} = gen_udp:open(6000),
    loop().

loop() ->
    receive
Any ->
    io:format("received:~p~n", [Any]),
    loop()
    end.
```

Здесь нам понадобится два порта, один будет широковещать, а остальные прослушивать. Мы выбрали 5010 порт для отправки широковещательных запросов и 6000 для прослушивания широковещательного траффика(Эти два числа не имеют значения; я просто выбрал два свободных порта на моей системе).

Открытие 5010 порта производит только тот процесс, который производит широковещательную рассылку, а все остальные машины в сети вызывают broadcast:listen(), для открытия 6000 порта и прослушивания широковещательных сообщений.

broadcast:send(IoList) посылает всем машинам в локальной сети IoList широковещательным сообщением.

Примечание: что бы это заработало, имя сетевого интерфейса должно быть корректным, и в локальной сети должно поддерживаться широковещание. На моем iMac, например, я использую имя "en0"вместо "eth0". Заметим так же, что если хост, который производит широковещание и хост, который прослушивает UDP широковещательный траффик находятся в разных подсетях, то второй вряд ли услышит первого, поскольку по умолчанию маршрутизаторы не пропускают широковещательные сообщения за пределы подсети.

14.7 SHOUTcast сервер

В завершении этого раздела, мы используем новоприобретенные навыки в сокет–программировании для написания SHOUTcast сервера. SHOUTcast —это протокол, который разработали в Nullsoft, для потокового вещания аудио данных.[6]. SHOUTcast отправляет MP3 или AAC закодированные аудиоданные, используя HTTP, как транспортный протокол.

Что бы увидеть, как это работает, вначале мы посмотрим SHOUTcast протокол. Затем мы посмотрим на общие структуры серверов, и закончим написанием кода.

14.7.1 SHOUTcast протокол

протокол SHOUTcast довольно прост:

1. Вначале клиент(который может быть чем-то вроде XMMS, Winamp или iTunes) посылает HTTP запрос SHOUTcast серверу. Вот запрос, который XMMS генерирует, когда я запускаю мой SHOUTcast сервер дома:

```
GET / HTTP/1.1
Host: localhost
User-Agent: xmms/1.2.10
Icy-MetaData:1
```

2. Мой SHOUTcast сервер отвечает вот так:

```
ICY 200 OK
icy-notice1: <BR>This stream requires
<a href=http://www.winamp.com/>;Winamp</a><BR>
icy-notice2: Erlang Shoutcast server<BR>
icy-name: Erlang mix
icy-genre: Pop Top 40 Dance Rock
icy-url: http://localhost:3000
content-type: audio/mpeg
icy-pub: 1
icy-metaint: 24576
icy-br: 96
... data ...
```

3. Теперь SHOUTcast сервер посылает непрерывный поток данных. данные имеют следующую структуру:
   H_0 F H F H F H ...[7]
   F — это блок MP3 аудио данных, который должен быть ровно 24 576 байов(число получено из icy-metaint параметра). H — это блок метаданных. Первый байт блока H есть целое число K, 16*K — это длина блока метаданных(без учета первого байта). Блок метаданных содержит строку вида: StreamTitle=' ...';SreamUrl=' ...'; Если длина этой строки не кратна 16, то оставшиеся байты будут забиты нулями.

   Заметим так же, что наикратчайший блок метаданных выглядит так <<0>>. Тоесть один байт длины с нулями.

[6] http://www.shoutcast.com/

[7] От автора перевода: В оригинальном тексте не упоминается про блок H_0(хотя в коде он есть), но для лучшего восприятия я решил его написать. этот блок генерируется единожды, когда клиент отправляет запрос на получение данных. H_0 генерируется функцией responce().

14.7.2 Как SHOUTcast сервер работает

Для создания сервера, мы будем соблюдать следующие детали:

1. Сделаем плейлист. Наш сервер будет использовать файл с ID3 тэгами, который мы создали в Главе 13.2. Песни будут выбираться случайным образом.

2. Сделаем параллельный сервер, таким образом мы сможем обслуживать несколько потоков параллельно. Мы сделаем это, используя технику, которая описана в 14.1(Параллельный сервер, на странице 254).

3. Из каждого файла мы будем отправлять только аудиоданные, без ID3 тэгов [8].

   Для удаления ID3 тэгов, мы будем использовать код из id3_tag_length; этот код использует код, разработанный на странице 232, код из раздела 5.3, Поиск и синхронизация фреймов в MPEG данных, на странице 92. Этот код не будет показан здесь.

14.7.3 Псведокод для SHOUTcast сервера

Перед тем,как мы увидим финальную программу, давайте взглянем на общий вид программы, опустив некоторые детали:

```
start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, ..),
    %% создание музыкального сервера -- он просто знает обо всей нашей музыке.
    PidSongServer = spawn(fun() -> songs() end),
    spawn(fun() -> par_connect(Listen, PidSongServer) end).


%% порождение одного из процессов на ожидание соединения
par_connect(Listen, PidSongServer) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    %% когда accept возвращает управление, мы порождаем
    %% новый процесс для ожидания соединения.
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},
{active, true}]),
    %% deal with the request
    get_request(Socket, PidSongServer, []).

%%ожидает TCP соединение
get_request(Socket, PidSongServer, L) ->
    receive
{tcp, Socket, Bin} ->
    ... Bin содержит клиентский запрос
    ... если запрос фрагментирован, то мы вызываем loop снова ...
    ... иначе мы вызываем
    ... got_request(Data, Socket, PidSongServer)
{tcp_closed, Socket} ->
    ... это происходит, если клиент обрывает соединение
    ... до того, как успел послать запрос (маловероятно)
    end.

%% мы получили запрос -- отправим ответ
```

[8] Непонятно, является ли это правильной стратегией. Аудио кодировщики предполагают перескакивание очень плохих данных, поэтому, мы можем отправлять ID3 тэги вместе с аудио-данными. Но на деле, кажется, что программа работает лучше, если мы удалим ID3 тэги.

```
got_request(Data, Socket, PidSongServer) ->
    .. data - это клиентский запрос ...
    .. анализируем его ...
    .. мы будем всегда обслуживать запрос ..
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer).

%% будем проигрывать песни, пока клиент не отсоединится.
play_songs(Socket, PidSongServer) ->
    ... PidSongServer хранит список всех MP3 данных
    Song = rpc(PidSongServer, random_song),
    ... Song - это случайная песня ...
    Header = make_header(Song),
    ... создание блока метаданных ...
    {ok, S} = file:open(File, [read,binary,raw]),
    send_file(1, S, Header, 1, Socket),
    file:close(S),
    play_songs(Socket, PidSongServer).
    send_file(K, S, Header, OffSet, Socket) ->
    ... отправка файла клиенту фрагментами ...
    ... эта функция возвращает управление, когда файл отправлен ...
    ... если произошла ошибка, при записи в сокет
    ... это означает, что клиент отсоединился.
```

Если вы посмотрите на реальный код, то вы увидите, что детали слегка отличаются, но идея так же самая. Вот полный листинг:

```
-module(shout).

%% в одном окне > shout:start()
%% в дургих окнах xmms http://localhost:3000/stream

-export([start/0]).
-import(lists, [map/2, reverse/1]).

-define(CHUNKSIZE, 24576).

start() ->
    spawn(fun() ->
    start_parallel_server(3000),
    %% теперь усыпляемся, посколько если это не сделать,
    %% то прослушиваемый сокет будет закрыт.
    lib_misc:sleep(infinity)
  end).


start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, [binary, {packet, 0},
{reuseaddr, true},
{active, true}]),
    PidSongServer = spawn(fun() -> songs() end),
    spawn(fun() -> par_connect(Listen, PidSongServer) end).


par_connect(Listen, PidSongServer) ->
```

```
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},{active, true}]),
    get_request(Socket, PidSongServer, []).
    get_request(Socket, PidSongServer, L) ->
    receive
{tcp, Socket, Bin} ->
    L1 = L ++ binary_to_list(Bin),
    %% split checks if the header is complete
    case split(L1, []) of
more ->
    %% заголовок собран не полностью, нужно больше данных.
    get_request(Socket, PidSongServer, L1);
{Request, _Rest} ->
    %% заголовок собран полностью
    got_request_from_client(Request, Socket, PidSongServer)
    end;
{tcp_closed, Socket} ->
    void;
_Any ->
    %% пропустим это
    get_request(Socket, PidSongServer, L)
    end.


split("\r\n\r\n" ++ T, L) -> {reverse(L), T};
split([H|T], L)           -> split(T, [H|L]);
split([], _)              -> more.


got_request_from_client(Request, Socket, PidSongServer) ->
    Cmds = string:tokens(Request, "\r\n" ),
    Cmds1 = map(fun(I) -> string:tokens(I, " " ) end, Cmds),
    is_request_for_stream(Cmds1),
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer, <<>>).


play_songs(Socket, PidSongServer, SoFar) ->
    Song = rpc(PidSongServer, random_song),
    {File,PrintStr,Header} = unpack_song_descriptor(Song),
    case id3_tag_lengths:file(File) of
error ->
    play_songs(Socket, PidSongServer, SoFar);
{Start, Stop} ->
    io:format("Playing:~p~n" ,[PrintStr]),
    {ok, S} = file:open(File, [read,binary,raw]),
    SoFar1 = send_file(S, {0,Header}, Start, Stop, Socket, SoFar),
    file:close(S),
    play_songs(Socket, PidSongServer, SoFar1)
    end.



send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
    %% OffSet = первый байт аудиоданных.
```

```
    %% Stop = последний байт аудиоданных.
    Need = ?CHUNKSIZE - size(SoFar),
    Last = OffSet + Need,
    if
Last >= Stop ->
    %% даже если мы и дочитаем файл до конца, то мы не
    %% наберем байтов до 24576, поэтому вычитываем, что есть
    %% и возвращаемся в play_songs
    Max = Stop - OffSet,
    {ok, Bin} = file:pread(S, OffSet, Max),
    list_to_binary([SoFar, Bin]);
true ->
    {ok, Bin} = file:pread(S, OffSet, Need),
    write_data(Socket, SoFar, Bin, Header),
    send_file(S, bump(Header),
    OffSet + Need, Stop, Socket, <<>>)
    end.



write_data(Socket, B0, B1, Header) ->
    %% Проверим, действительно ли данные для отправки имеют нужную длину.
    %% это очень полезная проверка, которая проверяет нашу программу на корректность.
    case size(B0) + size(B1) of
?CHUNKSIZE ->
    case gen_tcp:send(Socket, [B0, B1, the_header(Header)]) of
ok ->
    true;
{error, closed} ->
    %% это происходит, если медиаплеер
    %% прерывает соединение.
    exit(playerClosed)
    end;
_Other ->
    %% не посылаем блок,а сигнализируем об ошибке.
    io:format("Block length Error: B0 = ~p b1=~p~n" ,
    [size(B0), size(B1)])
    end.



bump({K, H}) -> {K+1, H}.

the_header({K, H}) ->
    case K rem 5 of
0 -> H;
_ -> <<0>>
end.


is_request_for_stream(_) -> true.


response() ->
    ["ICY 200 OK\r\n" ,
```

```
    "icy-notice1: <BR>This stream requires" ,
    "<a href=\" http://www.winamp.com/\">Winamp</a><BR>\r\n" ,
    "icy-notice2: Erlang Shoutcast server<BR>\r\n" ,
    "icy-name: Erlang mix\r\n" ,
    "icy-genre: Pop Top 40 Dance Rock\r\n" ,
    "icy-url: http://localhost:3000\r\n" ,
    "content-type: audio/mpeg\r\n" ,
    "icy-pub: 1\r\n" ,
    "icy-metaint: " ,integer_to_list(?CHUNKSIZE),"\r\n" ,
    "icy-br: 96\r\n\r\n" ].


songs() ->
    {ok,[SongList]} = file:consult("mp3data" ),
    lib_misc:random_seed(),
    songs_loop(SongList).


songs_loop(SongList) ->
    receive
{From, random_song} ->
    I = random:uniform(length(SongList)),
    Song = lists:nth(I, SongList),
    From ! {self(), Song},
    songs_loop(SongList)
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
{Pid, Reply} ->
    Reply
    end.


unpack_song_descriptor({File, {_Tag,Info}}) ->
    PrintStr = list_to_binary(make_header1(Info)),
    L1 = ["StreamTitle='" ,PrintStr,
"';StreamUrl='http://localhost:3000';" ],
    %% io:format("L1=~p~n",[L1]),
    Bin = list_to_binary(L1),
    Nblocks = ((size(Bin) - 1) div 16) + 1,
    NPad = Nblocks*16 - size(Bin),
    Extra = lists:duplicate(NPad, 0),
    Header = list_to_binary([Nblocks, Bin, Extra]),
    %% Header это блок метаданных.
    {File, PrintStr, Header}.


make_header1([{track,_}|T]) ->
    make_header1(T);
make_header1([{Tag,X}|T]) ->
    [atom_to_list(Tag),": " ,X," " |make_header1(T)];
make_header1([]) ->
    [].
```

14.7.4 Запустим SHOUTcast сервер

Запустим сервер, и проверим, как он работает, нам необходимо выполнить три шага:

1. Создать плейлист.
2. Запустить сервер.
3. Настроить клиент для работы с сервером.

14.7.5 Создание плейлиста

Для создания плейлиста нам надо выполнить следующие шаги:

1. Перейти в каталог, где лежит модуль mp3_manager.erl[9]
2. Изменить путь в функции start1, которая находится в файле mp3_manager.erl, на путь, который указывает на каталог с MP3 файлами.
3. Скомпилировать mp3_manager, и набрать в оболочке mp3_manager:start1(). Мы должны увидеть что-то вроде этого:

```
1> c(mp3_manager).
{ok,mp3_manager}
2> mp3_manager:start1().
Dumping term to mp3data
ok
```

Если вам интересно, то вы можете взглянуть на файл mp3data, что бы увидеть результаты анализа.

14.7.6 Запуск SHOUTcast сервера

Запустим сервер из оболочки следующей командой:

```
1> shout:start().
...
```

14.7.7 Тестирование сервера

1. Запустим плеер и укажем в его найтройках адрес сервера: http://localhost:3000

   На моей системе я использовал XMMS, которого запустил следующей командой: xmms http://localhost:3000

   Примечание: если вы хотите подключиться к серверу с другой машины, вы должны указать IP адрес сервера. Например, что бы подключиться к серверу с моей Windows машины, на которой установлен winamp, я вызвал Play > URL меню в винампе и ввел адрес http://192.168.1.168:3000 в диалоговом окне Open URL

   на моем iMac я использовал iTunes,я вызвал Advanced > Open Stream меню и вписал в него предыдущий url.

2. Вы увидите диагностический вывод в окне, в котором запущен сервер.
3. Enjoy!

[9] тот самый модуль из предыдущей главы, где мы извлекали метаданные из MP3 файлов.

14.8 Копаем глубже

В этой главе мы рассмотрели наиболее часто используемые функции для манипулирования сокетами. Вы можете найти больше информации о socket API в документации, на страницах gen_tcp, gen_udp, и inet.