Trusted - доверенный

Untrusted - небезопасный

socket — сокет

prompt — подсказка

cookie — кука

firewall — межсетевой экран

reference — ссылка

proxy - посредник

Глава 10

Распределённое программирование

В этой главе мы введём понятие библиотек и примитивов Эрланга, которые мы будем использовать для написания распределённых эрланговых программ. *Распределённые программы* — это программы, которые созданы для работы на сети компьютеров и могут координировать свои действия только через передачу сообщений.

Есть ряд причин, по которым мы можем захотеть написать распределённую программу. Вот некоторые из них:

Производительность

Мы можем заставить наши программы работать быстрее, организовав работу разных частей программы параллельно на разных машинах.

Надёжность

Мы можем сделать системы устойчивыми к сбоям, спроектировав их для работы на нескольких машинах. Если одна машина выходит из строя, то мы можем продолжить работу на другой машине.

Масштабируемость

С ростом приложения мы рано или поздно исчерпаем возможности даже самой мощной машины. После этого нам придётся добавлять новые машины для наращивания вычислительной мощности. Добавление новой машины должно быть простой операцией, которая не требует больших изменений в архитектуре приложения.

Прирождённая распределённость

Многие приложения являются распределёнными по своей сути. Если мы пишем многопользовательскую игру или чат, разные пользователи будут разбросаны по всему

земному шару. Если в каком-то географическом месте у нас будет много пользователей, то нам захочется разместить вычислительные ресурсы рядом с ними.

Развлечение

Большинство интересных программ, которые я пишу — распределённые. Многие из них включают взаимодействие людей и машин по всему миру.

В этой главе мы поговорим о двух основных моделях распределённости:

- *Распределённый Эрланг*: обеспечивает метод для программирования приложений, которые работают на наборе сильно связанных компьютеров1 . В распределённом Эрланге программы пишутся так, чтобы работать на *узлах* Эрланга. Мы можем порождать процессы на любом узле и все примитивы передачи сообщений и обработки ошибок, о которых мы говорили в предыдущих главах, работают как для случая одиночного узла.

Распределённые эрланговые приложения работают в *доверенной* среде — т.к. любой узел может выполнить любую операцию на любом другом узле Эрланга, то подразумевается высокая степень доверия. Типично распределённые эрланговые приложения работают на кусках одной локальной сети за межсетевым экраном, хотя, конечно, они могут работать и в открытой сети.

- *Распределение на основе сокетов*: используя TCP/IP сокеты, можно писать распределённые приложения, которые работают в *небезопасной* среде. Программная модель менее мощная, по сравнению с распределённым Эрлангом, но более безопасная. В части 10.5 *Распределение на основе сокетов*, на стр. ____ мы увидим, как создавать приложения, используя простой распределённый механизм на основе сокетов.

Если вы подумаете о предыдущих главах, то вспомните, что основной строительный блок для наших программ — это процесс. Писать распределённые программы на Эрланге легко: всё, что нам надо — это порождать наши процессы на правильных машинах, а затем всё будет работать как и раньше.

Все мы привыкли к написанию последовательных программ. Написание распределённых программ обычно гораздо труднее. В этой главе мы посмотрим на ряд техник для написания простых распределённых программ. И хотя эти программы просты, они очень полезны.

А начнём мы с ряда маленьких примеров. Для них нам понадобится изучить только две вещи, а затем мы сможем создать нашу первую распределённую программу. Мы узнаем, как запускать узел Эрланга и как выполнять удалённый вызов процедуры на удалённом узле Эрланга.

Когда я разрабатываю распределённое приложение, я всегда работаю над программой в определённом порядке:

Я пишу и тестирую программу в обычной, нераспределённой сессии Эрланга. Это то, где мы были до текущего момента, так что здесь не возникнет никаких новых проблем.

Я тестирую программу на двух различных узлах Эрланга, работающих *на одном компьютере*.

Я тестирую программу на двух различных узлах Эрланга, работающих на *двух физически*

разделённых компьютерах, находящихся либо в одной локальной сети, либо где-то в Интернете.

Последний шаг может оказаться проблематичным. Если мы работаем на машинах в одном административном домене, то это редко бывает проблемой. Но когда вовлечённые узлы принадлежат машинам из разных доменов, мы можем столкнуться с проблемой связи и нам придётся обеспечить, чтобы настройки межсетевого экрана и настройки безопасности были корректными.

В следующих частях мы создадим простой сервер имён, пройдя эти шаги по-порядку. Более точно, мы сделаем следующее:

Напишем и протестируем сервер имён в обычной, нераспределённой эрланговой системе.

Протестируем сервер имён на двух узлах на одной машине.

Протестируем сервер имён на двух разных узлах на двух разных машинах в одной локальной сети.

Протестируем сервер имён на двух разных машинах, относящихся к двум разных доменам в двух разных странах.

10.1 Сервер имён

Сервер имён — это программа, которая, получив имя, возвращает значение, связанное с этим именем. Мы также можем менять значение, связанное с определённым именем.

Наш первый сервер имён чрезвычайно прост. Он не устойчив к сбоям, так что все данные, хранящиеся в нём будут потеряны при сбое. Цель этого упражнения не сделать надёжный сервер имён, а начать разбираться с техниками распределённого программирования.

Шаг 1: Простой сервер имён

Наш сервер имён kvs — это простой сервер вида ключ-значение. У него следующий интерфейс:

@spec kvs:start() -> true

Запускает сервер; создаёт сервер с зарегистрированным именем kvs.

@spec kvs:store(Key, Value) -> true

Связывает ключ и значение.

@spec kvs:lookup(Key) -> {ok, Value} | undefined

Ищет значения для ключа и возвращает {ok, Value}, если с ключом связано значение; в противном случае возвращает undefined.

Сервер ключ-значение реализуется посредством примитивов get и put для словаря процесса:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"DownloadHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"socketHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"distHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"/HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"kvsHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/kvs.erl"erl

```
-module(kvs).

-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->

kvs ! {self(), Q},

receive

{kvs, Reply} ->

Reply

end.

loop() ->

receive

{From, {store, Key, Value}} ->

put(Key, {ok, Value}),

From ! {kvs, true},

loop();

{From, {lookup, Key}} ->

From ! {kvs, get(Key)},

loop()

end.
```

Мы начнём с локального тестирования сервера, чтобы посмотреть — работает ли он

корректно:

1> kvs:start().

true

2> kvs:store({location, joe}, "Stockholm").

true

3> kvs:store(weather, raining).

true

4> kvs:lookup(weather).

{ok,raining}

5> kvs:lookup({location, joe}).

{ok,"Stockholm"}

6> kvs:lookup({location, jane}).

undefined

Пока что у нас никаких неприятных сюрпризов.

Шаг 2: Клиент на одном узле, сервер на другом узле, но на той же машине

Теперь мы запустим два узла Эрланга на *одном* компьютере. Чтобы сделать это нам надо открыть два терминальных окна и запустить две системы Эрланга.

Первое: запускаем терминальную оболочку2 и в ней запускаем распределённый узел Эрланга с именем gandalf. Затем запускаем сервер:

$ erl -sname gandalf

(gandalf@localhost) 1> kvs:start().

true

Замечание для Windows: в системе Windows имя может оказаться не localhost. Если оно не localhost, то вам придётся использовать имя, которое Windows вернёт вместо localhost, во всех последующих командах.

Аргумент -sname gandalf означает «запустить узел Эрланга с именем gandalf на локальной машине». Заметьте, как оболочка Эрланга пишет имя узла3 HYPERLINK ""NameHYPERLINK ""@HYPERLINK ""Host. Name и Host — это атомы, так что они должны быть в одинарных кавычках, если они содержат какие-либо не атомные символы. Эрланга перед командной подсказкой.

Второе: запускаем *вторую* терминальную сессию и запускаем узел Эрланга с именем bilbo. После этого мы можем вызывать функции из kvs, используя библиотечный модуль rpc. (Заметьте, что rpc — это стандартный модуль библиотеки Эрланга, а не то, что мы написали ранее).

$ erl -sname bilbo

(bilbo@localhost) 1> rpc:call(gandalf@localhost,

kvs,store, [weather, fine]).

true

(bilbo@localhost) 2> rpc:call(gandalf@localhost,

kvs,lookup,[weather]).

{ok,fine}

Возможно это и не выглядит так, как надо, но мы только что выполнили наше первое распределённое вычисление! Сервер работал на первом узле, а клиент — на втором.

Вызов для установки переменной weather был сделан на узле bilbo. Мы можем вернуться обратно на узел gandalf и проверить значение weather:

(gandalf@localhost)2> kvs:lookup(weather).

{ok,fine}

rpc:call(Node, Mod, Func, [Arg1, Arg2, ..., ArgN]) выполняет *удалённый вызов процедуры* на узле Node. Функция, которая вызывается — это Mod:Func(Arg1, Arg2, ..., ArgN).

Как мы можем видеть, программа работает как для случая нераспределённого Эрланга. Единственное отличие — это то, что клиент работает на одном узле, а сервер — на другом.

Следующий шаг — это запуск клиента и сервера на разных машинах.

Шаг 3: Клиент и сервер на разных машинах в одной локальной сети

Мы будем использовать два узла. Первый узел — это gandalf на машине doris.myerl.example.com и второй узел — это bilbo на машине george.myerl.example.com. Перед тем, как сделать это, мы откроем два терминальных окна на двух разных машинах. Назовём эти окна *doris* и *george*. Когда мы это сделаем, мы сможем выполнять команды на обеих машинах.

Шаг 1: запускаем узел Эрланга на машине doris:

doris $ erl -name gandalf -setcookie abc

(gandalf@doris.myerl.example.com) 1> kvs:start().

true

Шаг 2: запускаем узел Эрланга на машине george и посылаем несколько команд к gandalf:

george $ erl -name bilbo -setcookie abc

(bilbo@george.myerl.example.com) 1> rpc:call("mailto:gandalf@doris.myerl.example.com", kvs, store, [weather,cold]).

true

(bilbo@george.myerl.example.com) 2> rpc:call("mailto:gandalf@doris.myerl.example.com", kvs, lookup, [weather]).

{ok,cold}

Всё ведёт себя в точности также, как для случая двух разных узлов на одной машине.

Чтобы это заработало сейчас, вещи должны быть чуть более сложные, чем в случае, когда мы запускали два узла на одной машине. Сейчас нам надо сделать следующее:

Запустить Эрланг с параметром -name. Когда у нас два узла на одной машине мы используем «короткие» имена (это видно по признаку -sname), но если они в разных сетях, то надо использовать параметр -name.

Мы можем использовать -sname в случае, когда машины в одной подсети. Использование -sname — это единственный рабочий способ при отсутствии ДНС.

Убедиться, что на обоих узлах одинаковые куки (cookie). Именно поэтому оба узла были запущены с параметром командной строки -setcookie abc. (Мы поговорим о куках позднее в этой главе5)

Убедиться, что полное имя машин для узлов разрешается ДНС-ом. В моём случае доменное имя myerl.example.com полностью локальное для моей домашней сети и разрешается локально добавлением записи в файл /etc/hosts.

Убедиться, что на обеих системах одинаковые версии кода , который мы хотим выполнить. В нашем случае одинаковые версии кода для kvs должны быть доступны на обеих системах. Есть несколько способов достичь этого:

- Дома у меня есть два физически раздельных компьютера без совместно используемых файлов. Здесь я физически копирую kvs.erl на обе машины перед запуском.
- На моём рабочем компьютере у меня рабочая станция с разделяемым по NFS диском. Здесь я просто запускаю Эрланг в разделяемом каталоге с двух различных рабочих станций.
- Сконфигурировать сервер кода делать это. Я не буду объяснять здесь — как это делать. Гляньте на руководство к модулю erl_prim_loader.

- Использовать команду шелла nl(Mod). Она загружает модуль Mod на всех подсоединённых узлах.

Замечание: чтобы это работало, надо чтобы все узлы были подсоединены. Узлы соединяются, когда они пытаются получить доступ друг к другу. Это происходит, когда вы впервые вычисляете выражение, включающее удалённый узел. Простейший способ сделать это — выполнить net_adm:ping(Node) (см. руководство по net_adm за дальнейшими деталями).

Шаг 4: Клиент и сервер на разных машинах в Интернете

В принципе, это то же самое, что и шаг 3, но сейчас нам надо гораздо больше позаботиться о безопасности. Когда мы запускаем два узла в одной локальной сети, нам, возможно, не надо сильно волноваться по поводу безопасности. В большинстве организаций локальная сеть отделена от Интернета межсетевым экраном. За этим экраном мы вольны выбирать IP адреса совершенно наобум и конфигурировать наши машины сколь угодно криво.

Когда же мы подключаем несколько машин эрлангового кластера к Интернету, мы можем ожидать проблем от межсетевых экранов, которые не пропускают входящие соединения. Нам нужно правильно сконфигурировать наши межсетевые экраны для приёма входящих соединений. Общей рекомендации как это сделать, не существует, т. к. все экраны различны.

Чтобы подготовить вашу систему к распределённому Эрлангу вам нужно выполнить следующие шаги:

Убедиться, что порт 4369 открыт для TCP и UDP трафика. Этот порт используется программой epmd (сокращение от Erlang Port Mapper Daemon).

Выбрать порт или диапазон портов для использования в распределённом Эрланге и убедиться, что эти порты открыты. Если эти порты от Min до Max (используйте Min=Max, если хотите использовать только один порт), то запускайте Эрланг следующей командой:

$ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \

inet_dist_listen_max Max

10.2 Примитивы распределения

Центральная концепция в распределённом Эрланге — это узел. Узел — это самодостаточная система Эрланга, содержащая полную виртуальную машину со своим собственным адресным пространством и собственным набором процессов.

Доступ к одиночному узлу или набору узлов обезопасен посредством кук (cookie). У каждого узла своя кука и эти куки должны быть одинаковы для всех узлов, с которыми наш узел собирается общаться. Чтобы обеспечить это, все узлы в распределённой эрланговой системе должны быть запущены с одинаковыми куками или должны установить свои куки в одинаковое значение вызовом erlang:set_cookie.

Набор соединённых узлов, имеющих одинаковые куки, образует эрланговый кластер.

Следующие встроенные функции (BIF) используются для написания распределённых программ7 :

@spec spawn(Node, Fun) -> Pid

Работает в точности, как spawn(Fun), только новый процесс порождается на узле Node.

@spec spawn(Node, Mod, Func, ArgList) -> Pid

Работает в точности, как spawn(Mod, Func, ArgList), только новый процесс порождается на узле Node. spawn(Mod, Func, Args) создаёт новый процесс, который вычисляет apply(Mod, Func, Args). Он возвращает PID нового процесса.

Замечание: эта форма порождения более надёжна, чем spawn(Node, Fun). spawn(Node, Fun) может сломаться, если на распределённых узлах работают хотя бы малость отличающиеся версии соответствующего модуля.

@spec spawn_link(Node, Fun) -> Pid

Работает в точности, как spawn_link(Fun), только новый процесс порождается на узле Node.

@spec spawn_link(Node, Mod, Func, ArgList) -> Pid

Работает в точности, как spawn(Node, Mod, Func, ArgList), только новый процесс связывается с текущим процессом.

@spec disconnect_node(Node) -> bool() | ignored

Принудительно отсоединяет узел.

@spec monitor_node(Node, Flag) -> true

Если Flag имеет значение true, то включается мониторинг. Если Flag имеет значение false, то мониторинг выключается. При включенном мониторинге процессу, который выполнил эту функцию, посылаются сообщения {nodeup, Node} и {nodedown, Node} в случае, когда узел Node присоединяется или покидает набор подключенных узлов Эрланга.

@spec node() -> Node

Возвращает имя локального узла. Если узел не является распределённым, то возвращается nonode@nohost.

@spec node(Arg) -> Node

Возвращает узел, где находится Arg. Arg может быть PID, ссылка или порт. Если узел не является распределённым, то возвращается nonode@nohost.

@spec nodes() -> [Node]

Возвращает список всех других узлов в сети, с которыми мы соединены.

@spec is_alive() -> bool()

Возвращает true, если локальный узел жив и может быть частью распределённой системы. В противном случае возвращает false.

Дополнительно, для отправки сообщений к процессу, зарегистрированному локально в наборе распределённых узлов Эрланга может использоваться send. Синтакс этого следующий:

{RegName, Node} ! Msg

посылает сообщение Msg к зарегистрированному на узле Node процессу RegName.

Пример удалённого порождения

Как простой пример, мы можем показать — как порождать процесс на удалённом узле. Начнём со следующей программы:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl"distHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl"demoHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/dist_demo.erl"erl

```
-module(dist_demo).

-export([rpc/4, start/1]).

start(Node) ->

spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->

Pid ! {rpc, self(), M, F, A},

receive

{Pid, Response} ->

Response

end.

loop() ->

receive

{rpc, Pid, M, F, A} ->

Pid ! {self(), (catch apply(M, F, A))},
```

```
loop()

end.
```

Затем мы запускаем два узла, причём оба узла должны быть способны загрузить этот код. Если оба узла запущены на одной машине, тогда это не проблема. Мы просто запускаем два Эрланга из одного и того же каталога. Если же узлы находятся на физически распределённых машинах с разными файловыми системами, то программу надо скопировать на все машины и скомпилировать перед запуском обоих узлов (ну или можно скопировать .beam файл на все машины). В этом примере я полагаю, что это уже сделано.

На машине doris мы запускаем узел под названием gandalf:

```
doris $ erl -name gandalf -setcookie abc

(gandalf@doris.myerl.example.com) 1>
```

А на машине george мы запускаем узел под названием bilbo, помня об использовании тех же кук:

```
george $ erl -name bilbo -setcookie abc

(bilbo@george.myerl.example.com) 1>
```

Теперь (на bilbo) мы можем породить процесс на удалённом узле (gandalf):

```
(bilbo@george.myerl.example.com) 1> Pid =

dist_demo:start('gandalf@doris.myerl.example.com').

<5094.40.0>
```

Pid — это идентификатор процесса на *удалённом узле* и теперь мы можем вызвать dist_demo:rpc/4 для выполнения удалённого вызова процедур на удалённом узле:

```
(bilbo@george.myerl.example.com)2> dist_demo:rpc(Pid, erlang, node, []).

'gandalf@doris.myerl.example.com'
```

Это выполняет erlang:node() *на удалённом узле* и возвращает значение.

10.3 Библиотеки для распределённого программирования

Предыдущая часть показала встроенные функции, которые мы можем использовать для написания распределённых программ. Фактически, большинство программистов на Эрланге никогда не используют эти функции. Вместо этого они используют ряд мощных библиотек для распределения. Библиотеки написаны с использованием этих функций, но они скрывают большинство сложностей от программиста.

Два модуля из стандартной поставки покрывают большинство нужд:

- rpc обеспечивает ряд сервисов удалённого вызова процедур;
- в global есть функции для регистрации имён и блокировок в распределённой системе и для поддержки полностью соединённой сети.

Читайте руководство по RPC

Модуль rpc — это настоящий рог изобилия функциональности

Одна наиболее полезная функция из модуля rpc — следующая:

call(Node, Mod, Function, Args) -> Result l {badrpc, Reason}

Она выполняет apply(Mod, Function, Args) на узле Node и возвращает результат Result или {badrpc, Reason}, в случае неуспеха.

10.4 Защита с помощью кук

Для того, чтобы два распределённых узла Эрланга могли общаться между собой, им надо иметь одинаковые *куки* (*magic cookie*). Мы можем установить куки тремя способами:

- *Способ 1*: сохранить одинаковые куки в файле $HOME/.erlang.cookie. Этот файл содержит строку случайных данных и создаётся автоматически, когда Эрланг запускается на вашей машине в первый раз.

Этот файл можно скопировать на все машины, которые будут участвовать в сеансе распределённого Эрланга. Или же мы можем явно установить значение. К примеру, на Linux мы можем выполнить следующие команды:

$ cd

$ cat > .erlang.cookie

AFRTY12ESS3412735ASDF12378

$ chmod 400 .erlang.cookie

Команда chmod делает файл .erlang.cookie доступным только владельцу файла.

- *Способ 2*: когда запускается Эрланг мы можем использовать параметр командной строки -setcookie C, чтобы установить значение куки в C. Пример:

$ erl -setcookie AFRTY12ESS3412735ASDF12378 ...

- *Способ 3*: встроенная функция erlang:set_cookie(node(), C) устанавливает куку на локальном узле в атом C.

Замечание: если ваше окружение небезопасно, то способы 1 и 3 предпочтительнее по сравнению со способом 2, т.к. на UNIX любой может узнать вашу куку, используя команду ps.

Если вам любопытно, то куки никогда не передаются по сети в открытом виде. Куки используются только для первоначальной аутентификации сеанса. Сеансы распределённого

Эрланга не шифруются, но они могут быть запущены поверх шифрованных каналов. (Поищите на Гугле более современную информацию из списка рассылки Эрланга).

10.5 Распределение на основе сокетов

В этой части мы напишем простую программу, используя распределение, основанное на сокетах. Как мы уже видели, распределённый Эрланг хорош для написания кластерных приложений, где вы можете доверять всем присутствующим, но не очень подходит для открытой среды, где нельзя доверять первому встречному.

Основная проблема с распределённым Эрлангом — это то, что клиент может породить *любой* процесс на серверной машине. Так что, для полного разрушения файловой системы всё, что вам надо сделать — это выполнить следующее:

```
rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *" ])
```

Распределённый Эрланг хорош в случае, когда все машины ваши и вы хотите управлять ими из одного места. Однако эта модель вычислений не подходит для случая, когда разные машины принадлежат разным людям и они хотят иметь контроль над тем, какие программы будут выполняться на их машинах.

В таких обстоятельствах мы будем использовать ограниченную версию порождения, когда у владельца определённой машины есть контроль над тем, что запускается на его машине.

lib_chan

lib_chan — это модуль, который позволяет пользователю явно управлять тем, какие процессы порождаются на его машине. Реализация lib_chan достаточно сложна, так что я не буду излагать её здесь. Вы можете найти её в Приложении D на стр. ____. Интерфейс у неё следующий:

```
@spec start_server() -> true
```

Запускает сервер на локальной машине. Поведение сервера определяется содержимым файла $HOME/.erlang/lib_chan.conf.

```
@spec start_server(Conf) -> true
```

Запускает сервер на локальной машине. Поведение сервера определяется содержимым файла Conf.

В обоих случаях в файле конфигурации сервера находятся кортежи следующего вида:

```
{port, NNNN}
```

Запускает приём соединений на порту NNNN

```
{service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS}
```

Сервис S защищается паролем P. При запуске сервиса для обработки сообщений от клиента

создаётся процесс посредством порождения SomeMod:SomeFunc(MM, ArgsC, SomeArgsS). Здесь MM — это PID процесса-посредника, который используется для передачи сообщений клиенту, а параметр ArgC приходит из клиентского вызова на подключение (к серверу).

@spec connect(Host, Port, S, P, ArgsC) -> {ok, Pid} | {error, Why}

Пытается открыть порт Port на машине Host и затем пытается активировать сервис S, который защищён паролем P. Если пароль верный, то возвращается {ok, Pid}, где Pid — это идентификатор процесса-посредника используемого для передачи сообщений к серверу.

Когда соединение устанавливается из клиента посредством вызова connect/5, создаются два процесса-посредника: один на стороне клиента и один на стороне сервера. Эти процессы организуют преобразование сообщений Эрланга в пакеты данных TCP, перехватывая при этом выходы управляющих процессов и закрытие сокета.

Это объяснение может показаться сложным, но оно окажется гораздо более простым, когда мы начнём его использовать.

Далее идёт полный пример использования lib_chan совместно с описанным ранее сервисом kvs.

Код сервера

Для начала напишем файл конфигурации:

{port, 1234}.

{service, nameServer, password, "ABXy45" ,

mfa, mod_name_server, start_me_up, notUsed}.

Это означает, что мы собираемся предлагать сервис, называемый nameServer на порту 1234 нашей машины. Сервис защищается паролем ABXy45.

Когда с клиента устанавливается соединение посредством вызова

connect(Host, 1234, nameServer, "ABXy45", nil)

сервер порождает mod_name_server:startmeUp(MM, nil, notUsed) (так в pdf-оригинале), где MM — это PID процесса-посредника, который используется для общения с клиентом.

Важно: сейчас вы должны глазеть на предыдущую строку кода и пытаться понять — откуда же взялись аргументы для этого вызова:

- mod_name_server, start_me_up и notUsed взяты из файла конфигурации
- nil — это последний аргумент в вызове connect.

Модуль mod_name_server выглядит так:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"DownloadHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"socketHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"*HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"distHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"/HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"modHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"*HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"nameHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"serverHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/socket_dist/mod_name_server.erl"erl

```
-module(mod_name_server).

-export([start_me_up/3]).

start_me_up(MM, ArgsC, ArgS) ->

loop(MM).

loop(MM) ->

receive

{chan, MM, {store, K, V}} ->

kvs:store(K, V),

loop(MM);

{chan, MM, {lookup, K}} ->

MM ! {send, kvs:lookup(K)},

loop(MM);

{chan_closed, MM} ->

true

end.
```

mod_name_server работает по следующему протоколу:

- если клиент посылает серверу сообщение {send, X}, то оно появится в mod_name_server как сообщение вида {chan, MM, X} (MM — это PID серверного процесса-посредника).
- если клиент завершается или сокет, используемый для связи, закрывается по какой-либо причине, то сервер получает сообщение вида {chan_closed, MM}.

- если сервер хочет послать сообщение X клиенту, он делает это посредством вызова MM ! {send, X}.
- если сервер хочет закрыть соединение явно, он делает это, выполняя MM ! close.

Этот протокол — протокол посредника, которому подчиняются как клиентский, так и серверный код. Код сокета для посредника объясняется более подробно в части D.2, *lib_chan_mm: Посредник*, на стр. ____.

Чтобы протестировать этот код, мы сначала убедимся, что он работает правильно на одной машине.

Запускаем сервер имён (и модуль kvs):

1> kvs:start().

true

2> lib_chan:start_server().

Starting a port server on 1234...

true

После этого мы можем запустить второй сеанс Эрланга и протестировать всё это со стороны клиента:

1> {ok, Pid} = lib_chan:connect("localhost", 1234, nameServer,

"ABXy45", "").

{ok, <0.43.0>}

2> lib_chan:cast(Pid, {store, joe, "writing a book"}).

{send,{store,joe,"writing a book"}}

3> lib_chan:rpc(Pid, {lookup, joe}).

{ok,"writing a book"}

4> lib_chan:rpc(Pid, {lookup, jim}).

undefined

Проверив, что это работает на одной машине, мы проходим те же описанные ранее шаги и выполняем подобные тесты на двух физически разделённых машинах.

Заметьте, что в этом случае содержимое конфигурационного файла определяется владельцем удалённой машины. Файл конфигурации указывает какие приложения разрешены на этой машине и какой порт должен использоваться для связи с этими приложениями.

Используя что-нибудь вроде ssh.

И одинаковая версия Эрланга. Если вы не сделаете этого, вы получите странные и страшные ошибки.