## 1.3 Многопроцессорное и распределенное взаимодействие

### 1.3.1 Общая характеристика процессов в Erlang

Erlang имеет модель параллелизма, которая основана на использовании процессов, причем легковесных. Процесс в Erlang не имеет ничего общего с тяжеловесным процессом или потоком операционной системы. Процессы на Erlang могут выполнять параллельно заданные функции, при этом процессы работают внутри виртуальной машины Erlang. Создать процесс очень просто и это не занимает много времени. Управлением процессов занимается планировщик внутри виртуальной машины Erlang. При этом планировщик может планировать сотни тысяч работающих процессов, в то время как диспетчер ОС работает лишь с сотнями тяжеловесных процессов. Процесс в Erlang может быть запущен на другой машине и в этом случае обмен с ним производится по сети без участия пользователя.

Каждый процесс идентифицируется Pid – идентификатором процесса. Это – специальный тип в Erlang.

Процесс в Erlang не имеет общих данных с другими процессами (кроме параметров запускаемой функции). Он может взаимодействовать с другими процессам лишь через сообщения.

Максимальное число процессов может быть задано из командной строки при запуске интерпретатора (машины) Erlang:

**erl –P** <число процессов> (для имеющейся сборки допустимы 1024-134217727 процессов, число зависит от сборки и от ОС)

Число физических ядер или процессоров, которое будет использовано в системе для планирования на них всех процессов Erlang, задается

**erl -S n1:n2** , где n1 – общее число ядер в системе, n2 – число активных ядер для работы процессов Erlang. При этом на каждом ядре будет запущен свой планировщик, планирующий легковесные процессы виртуальной машины Erlang на процессы в ОС.

### 1.3.2 Создание процесса

Процесс всегда создается как сущность, параллельно выполняющая некоторую функцию. Процессы порождает в любом месте кода, где разрешено выражение, функция spawn:

- Простой способ:
  spawn/1 (функтор)
  в коде:

```
spawn(fun ()-> io:format("i'm a lightweight process~n")
end).
```

как функция, которая вызывается как обычно и передает параметры процессу:

```
processNN(N) ->  spawn(fun ()-> N*N end).
```

В этом случае N*N будет вычислено параллельно

- Общий случай:
  spawn/3 (модуль, функция, [параметры])
```
processPrimesInParallel(N)-> io:format("primes ~p ~n",
[primes (N)]).
start() ->
spawn(mod, processPrimesInParallel, [10]).
```

- Запуск на другом узле
  spawn(Node, Module, Function, Args) - будет рассмотрен далее.

Во всех случаях spawn возвращает объект типа Pid(), который может быть использован для управления процессом и для взаимодействия с ним. Поэтому пример с N*N выше не вернет результат выражения – он вернет Pid, по которому мы можем получить результат только через работу с сообщениями.

### 1.3.3 Отправка и получение сообщений

Процессы в Erlang слабосвязанные. Все обмены производятся через сообщения, которые также сопоставляются по шаблонам. Операции приема и получения сообщений – это часть языка.

**Отправка** сообщения осуществляется с использованием операции !:

```
Pid ! Message
```

где Pid – это либо pid процесса, ранее возвращенный функцией spawn, либо атом, который можно зарегистрировать, дав псевдоним pid-у функцией register(Name, Pid). Можно послать сообщение самому себе, указав в качестве pid self().

Следует учесть, что отправка сообщения процессу означает постановку данного сообщения в очередь к посылаемому процессу и не происходит синхронно. Процессы не должны синхронизироваться через отправку, а только через получение.

**Получение** сообщения осуществляется оператором receive … end:

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
     end
```

Здесь Pattern1…PatternN – выражения, с которыми необходимо осуществить сопоставление, каждое такое выражение – ключ к выполнению соответвующего Body, это выражение будет выполнено при получении сообщения, если сопоставление выполнится. Также дополнительное можно надожить ограничения GuardSeq, как это делается при определении функций.

Получение сообщения всегда блокирующая операция, пока в очередь сообщения потока не будет положено соответвущее сообщение, блок receive…end будет блокирован.

Необязательный блок after необходим, если требуется выйти из режима ожидания сообщения по тайм-ауту. ExprT – выражение или константа, вычисление которого должно вернуть значение таймаута в миллисекундах.

Для примера разработаем два процесса, которые играют в пинг-понг.

**Пинг-понг:**

- Один процесс шлет другому Ping, и ждет ответа
- Другой получает Ping, и шлет Pong в ответ.

Процессы запускаются параллельно, причем процесс ping должен знать pid процесса pong, чтобы общаться с ним:

```
main() -> spawn(fun() -> ping(spawn(fun()-> pong() end)) end).
```

Процесс ping (получает pid процесса pong):

```
ping(PidPong)->
     io:format("I'm ping, pids are : ~w ~w ~n ",[PidPong, self()]),
     PidPong ! {ping, self()},
```

```
        io:format("waiting for pong...~n"),
        receive
              pong ->    io:format("PONG received~n")
        end.
```

Процесс ping шлет сообщение процессу pong, составляющее кортеж { ping, Pid }, при этом ping – это атом, идентифицирует сообщение и pid, в который будет передан свой pid (чтобы процесс pong мог ответить процессу обратно).

После чего процесс ping ждет сообщение pong от процесса pong и завершает работу.

Процесс pong:

```
pong()->
        io:format("I'm pong, my pid is : ~w ~n ",[self()]),
receive
        {ping , PidPing} ->
        io:format("PING received, sending pong~n "),
        PidPing ! pong
        end.
```

Pong ждет сообщение {ping, Pid}, при получении которого в результате сопоставления получим pid процесса ping, после чего на этот pid посылается ответное сообщение.

Работа:

```
1> proc:main().
I'm ping, pids are : <0.68.0> <0.67.0>
 I'm pong, my pid is : <0.68.0>
 waiting for pong...
PING received, sending pong
 <0.67.0>
PONG received
```

Рассмотрим модификацию примера с обходом каталогов. Для проверки работы большого числа процессов сделаем так, чтобы **каждый каталог обходился в своем процессе**.

```
recurserEx(Dir)->
```

```
    {ok, Filenames}=file:list_dir(Dir),
    processOneEx(Dir,Filenames).


processOneEx(Parent, [First|Next])->
    FullName=Parent++"/"++First,
    IsDir=filelib:is_dir(FullName),
    if
        IsDir==true -> erlang:spawn(fun()-> recurserEx(FullName)
end);
        IsDir==false -> io:format(FullName++"~n")
    end,
    processOneEx(Parent,Next);
processOneEx(_Parent,[])->ok.
```

Как мы увидим дальше, при создании большого числа процессов Elang выдерживает такой стресс-тест.

Еще один пример – **реализация модели файловой системы будущего на процессах**. Имеется множество файлов на диске, и столько же процессов. Спросив процесс с именем файла, можно получить содержимое этого файла.

Для решения задачи необходимо использовать групповые рассылки, т.е. посылку сообщений от одного процесса многим. Можно запустить процессы функцией spawn(), при создании сохранить их pid-ы в списке, а для рассылки проходить по списку и отсылать сообщение соответственному процессу. Рекомендуется использовать библиотеки для работы с группами процессов, например, pg (process group)[1].

Модифицированная функция рекурсивного обхода:

```
recurserM(Dir)->
    {ok, Filenames}=file:list_dir(Dir),
    processOneM(Dir,Filenames).
processOneM(Parent, [First|Next])->
    FullName=Parent++"/"++First,
    IsDir=filelib:is_dir(FullName),
    if
```

[1] Ранее библиотека входила в поставку Erlang, потом была оттуда убрана. Скачать можно, например, с https://github.com/simplegeo/erlang/blob/master/lib/stdlib/src/pg.erl

```
            IsDir==true ->  recurserM(FullName);
            IsDir==false -> io:format(FullName++"~n"),
erlang:spawn(fun() ->processBody(FullName) end)
      end,
      processOneM(Parent,Next);
processOneM(_Parent,[])->ok.
```

При обходе списка файлов в каталоге определяется файл ли это, и если файл, то создается процесс для обработки для этого файла:

```
processBody(FileName) ->
      pg:join(grp,self()),
      receive
            {pg_message, From, PgName, {getfile, Name}} ->
                  Test = string:equal(FileName, Name),
                  if Test == true -> {ok, Bin} =
file:read_file(FileName), From ! Bin, io:format(Bin);
                  Test == false ->ok
                  end
end,
processBody(FileName).
```

Данный код постоянно обрабатывает приходящие сообщения (обратите внимание, он зациклен). При старте процесс добавляется в группу процессов grp, далее ожидает сообщения. {pg_message, From, PgName, {getfile, Name}} – форма получаемых сообщений библиотеки pg, в данном случае {getfile, Name} – это данные сообщения (атом getfile и параметр, какое имя запрошено). Если имя процесса совпадает с именем файла, процесс читает файл с использованием модуля file и возвращает результат запросившему процессу.

Запускающая все это функция выглядит так:

```
startFS()->
      pg:create(grp), %%  создаем группу процессов
      recurserM("."),
      pg:send(grp, {getfile,"./first/src/proc.erl"}), %%посылка группе
      receive
            Msg->
```

```
                io:format("~w", Msg) %%выводим результат
end.
```

Замечание: она не совсем корректна, необходимо уведомить главный процесс, когда все процессы будут созданы и ожидать сообщений, после чего уже передавать запрос на получение данных файла.

### 1.3.4 Связи процессов и слежение за состоянием процесса

В Erlang возможно построить иерархию процессов, контролируя тем самым ошибки. Предположим, что необходимо узнать, что запущенный нами процесс отработал и как он отработал.

**Связь процессов (link)** –двухсторонняя, при ошибке в одном из процессов возможны 2 варианта:

1. автоматически другому процессу отправляется сообщение {'EXIT', From, Reason} (для этого необходимо вызвать process_flag(trap_exit, true) в процессе, который хочет обрабатывать сообщения;
2. оба процесса аварийно завершают работу (поведение по умолчанию).

Создать связь можно, создав процесс с использованием spawn_link (параметры аналогично spawn), или это можно проконтролировать позже с использованием link(pid)/unlink(pid). При этом вызов spawn_link – атомарный, т.е. не может быть такого варианта, когда процесс будет создан spawn, потом упадет, потом вызовется link.

Пример связи:

```
proc1() -> 1/0. % Деление на ноль
proc2() -> process_flag(trap_exit, true)
,spawn_link(fun()-> proc1() end),
receive
{'EXIT', From, Reason}
-> io:format("~w",[Reason])
end.
```

Процесс proc2 создает proc1 и ждет сообщения об ошибке, proc2 делит на 0, падает и посылает сообщение процессу proc1. Вывод:

```
{badarith,[{proc,proc1,0,[{file,[47,…,108]},{line,137}]}]}ok
```

Для завершения процесса из кода можно использовать функцию exit(атом), где атом – normal либо любой другой ошибочной.

Другой способ слежения за процессом – **мониторы**. Процесс монитор, в отличие от связи, работает в одну сторону. Кроме этого, может быть сколько угодно мониторов для каждого процесса.

Монитор создается либо при создании процесса spawn_monitor, либо monitor(process, Pid), где process- именно такой атом. В любом случае, после завершения процесса будет послано сообщение всем мониторам {'DOWN', Ref, process, Pid2, Reason}. Пример:

```
proc1() -> ok.
proc2() -> spawn_monitor (fun()-> proc1() end),
receive
Reason
-> io:format("~w",[Reason])
end.
```

Результат {'DOWN',#Ref<0.0.3.435>,process,<0.367.0>,normal}ok,

а при

proc1() -> 1/0.

```
{'DOWN',#Ref<0.0.3.553>,process,<0.404.0>,{badarith,[{proc,proc1,0,[{fil
e,[47,…,108]},{line,137}]}]}}ok
```

### 1.3.5 Словари процессов

Известно, что процесс выполняет функцию, а переменные в Erlang не переприсваиваются. Как обеспечить, например, запоминание информации между вызовами функции процесса? В Erlang были добавлены словари или ассоциативные массивы данных (аналог std:map в C++).

put(Key, Value) кладет в словарь данные Value по ключу Key для текущего процесса

Value=get(Key) по ключу возвращает значение, get_keys() возвращает все ключи, erase (Key) – удаляет значение по ключу.

Следует учесть, что использование словарей, строго говоря, нарушают функциональную парадигму(мы вводим меняющиеся переменные), и стоит использовать их, когда обычные средства (рекурсия с параметрами, сообщения в т.ч. самому себе) неприменимы.

### 1.3.6 Принципы параллельного программирования на Erlang

Создатели языка рекомендуют:

1. Больше вычислений, меньше сообщений. Сообщения – это рассылка данных (далее мы рассмотрим, что сообщение может идти по разным узлам и идти по сети), ожидания (блокировки), сопоставление. Это замедляет работу программы
2. Размер сообщений как можно меньше. Это позволит передавать меньше и меньше сопоставлять.
3. Не нужно вычислять в процессах маленькие функции. Хотя процессы очень просто создать, но не стоит вычислять простые вещи в процессах и отсылать результат, это увеличивает время работы.
4. Если можно выиграть в параллельности, то почему бы не попробовать. Сама структура яыка Erlang дает возможность писать параллельный код без общих данных.
5. Процессов в программе должно быть “Lagom” [2], это шведское слово означает ни много, ни мало, достаточно.

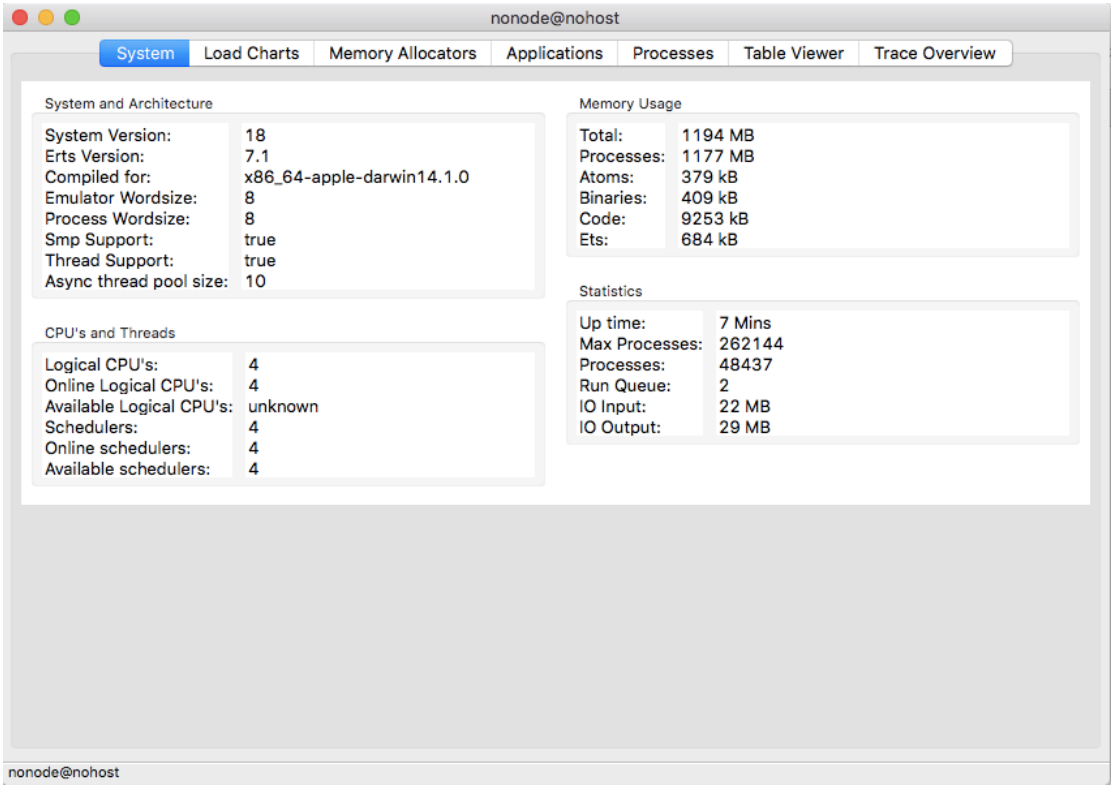
### 1.3.7 Анализ работы параллельных процессов в Erlang

В стандартном модуле erlang: имеются функции process_info(Pid), processes() и другие, позволяющие узнать информацию о текущих процессах в виртуальной машине Erlang. Также присутствует графическое средство, позволяющее мониторить все основные параметры, относящиеся к процессам, памяти и плинировщику. Средство запускается

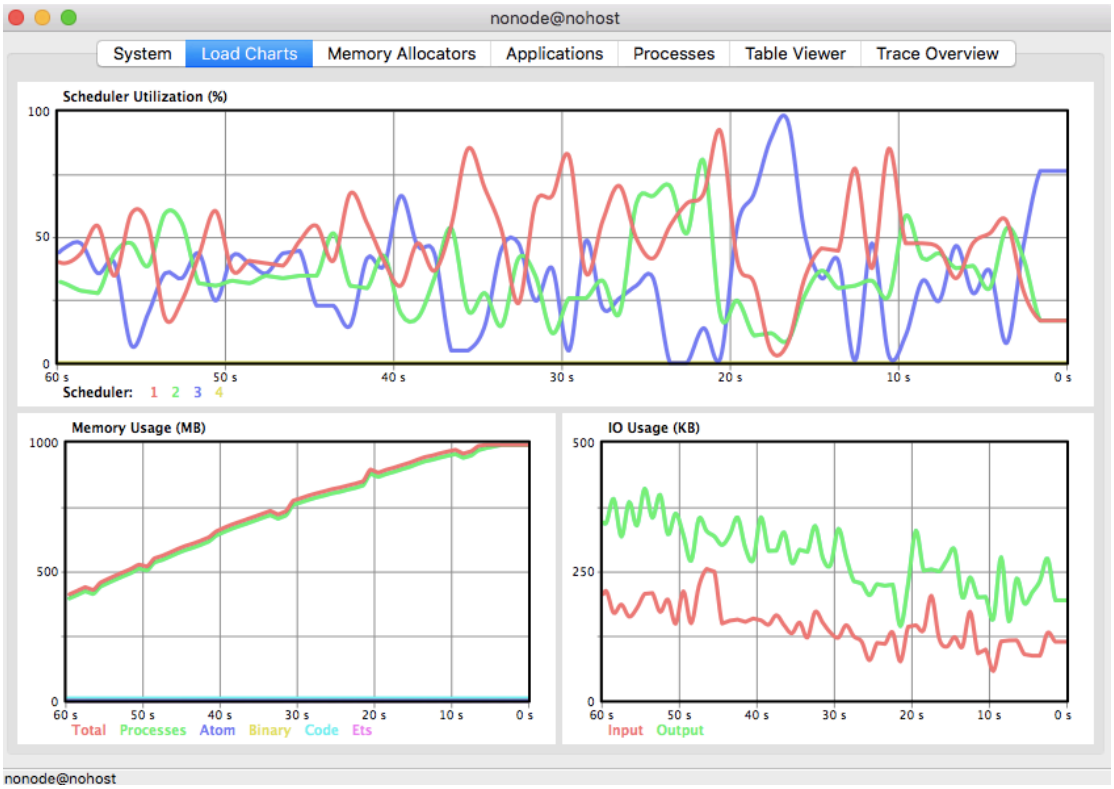
observer:start().

При этом средство работает в отдельном окне и позволяет запускать другие процессы. Для примера запустим пример с рекурсивным обходом дерева каталогов и созданием процесса для каждого каталога, выполнив вызов recurserEx(“/”).

[2] * lagom(шв.) The concept of lagom is similar to Russian expression normal'no (википедия)



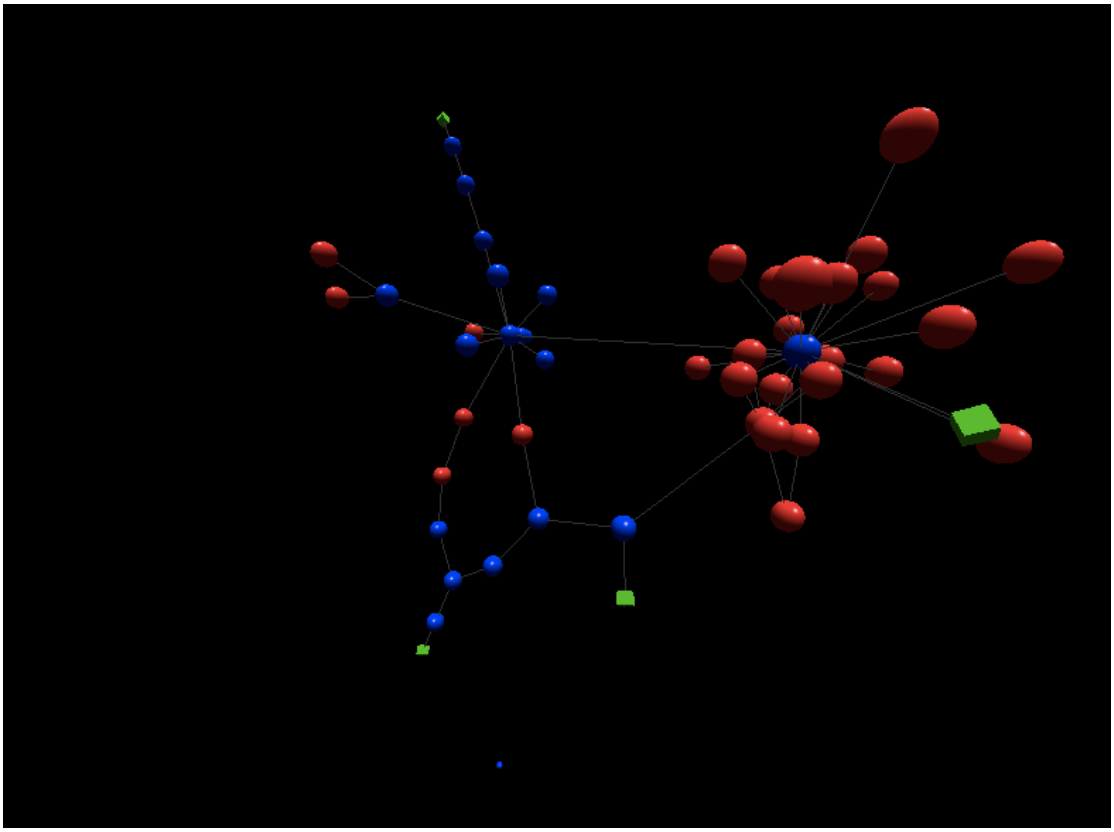

Как видно, показывается информация о доступных ядрах для планировщика, количество процессов и другая информация.




На вкладке с диаграммами загрузки видны использование доступных диспетчеров, использование диска и памяти. Также можно узнать информацию о процессах, вплоть до стека процесса и его мониторов.

Интересным средством является Ubigraph+erlubi[3], которая отрисовывает процессы в системе в виде трехмерного графа.

На рисунке показаны процессы в системе и пользовательские процессы при работе функции с обходом папок.



### 1.3.8 Распределенное программирование

Распределенное программирование – вид параллельного программирования, когда программа распределена по различным узлам, которые могут находиться удаленно, как правило при этом все взаимодействие производится по сети. Распределенное программирование на Erlang для программиста мало отличается от параллельного, мы имеем дело с процессами и сообщениями по pid-ам, но в случае распределенного программирования pid может указывать на процесс в другом узле.

Узел (Node) – сессия виртуальной машины, запущенная под своим именем. В Erlang узлы имеют тип атомов в ‘’, имя узла включает в себя логическое имя узла @ имя компьютера (или IP).

Имя узла задается при запуске erl с параметром –name (или в среде разработки через меню):

[3] https://github.com/krestenkrab/erlubi

erl -name machine1

Erlang/OTP 18 [erts-7.1] [source-2882b0c] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V7.1 (abort with ^G)

(machine1@sergeys-Air.Dlink)1>

Чтобы создать процесс на удаленном узле (предварительно его запустив), достаточно записать его адрес в качестве первого параметра к spawn/4. При этом модуль с запускаемой функцией должен быть также находиться на удаленном узле и быть скомпилированным. Пример – модифицированный запуск процессов для пинг-понга:

```
main() -> spawn(fun() -> ping(spawn('machine2@sergeys-MacBook-
Air.wifi.astu',proc, pong,[])) end).
```

Реализация процессов Ping и Pong не отличается, однако теперь процесс Pong запускается на другом узле machine2. Вывод:

<0.54.0>

I'm ping, pids are : <9857.44.0> <0.54.0>

waiting for pong...

I'm pong, my pid is : <9857.44.0>

PING received, sending pong

PONG received

Обратите внимание, pid-ы процессов выглядят немного по-другому: <0.54.0> - pid процесса Ping, он локальный и начинается с 0, <9857.44.0> - pid процесса Pong, он удаленный.

Функция nodes() выводит список известных узлов. Узел становится известным, как только успешно создается процесс на нем, либо на нашем узле какой-то узел создает процесс, либо он будет проверен командой net_adm:ping.

Первоначально этот список пуст, после запуска примера с ping-pong:

machine1@sergeys-Air.Dlink)5> nodes().

['machine2@sergeys-Air.Dlink']

На другом узле:

(machine2@sergeys-Air.Dlink)1> nodes().

['machine1@sergeys-Air.Dlink']

Функция node() возвращает текущий узел.

Однако, такой способ работы не совсем безопасен, ведь любой может запустить код на любом узле лишь по его имени. Поэтому введена система cookie – ключевых строк, которые идентифицируют узел. Для возможности запуска на данном узле какого-то кода необходимы, чтобы cookie значения узлов, которые запускают и выполняют код, были одинаковы.

Функция get_cookie() возвращает cookie.

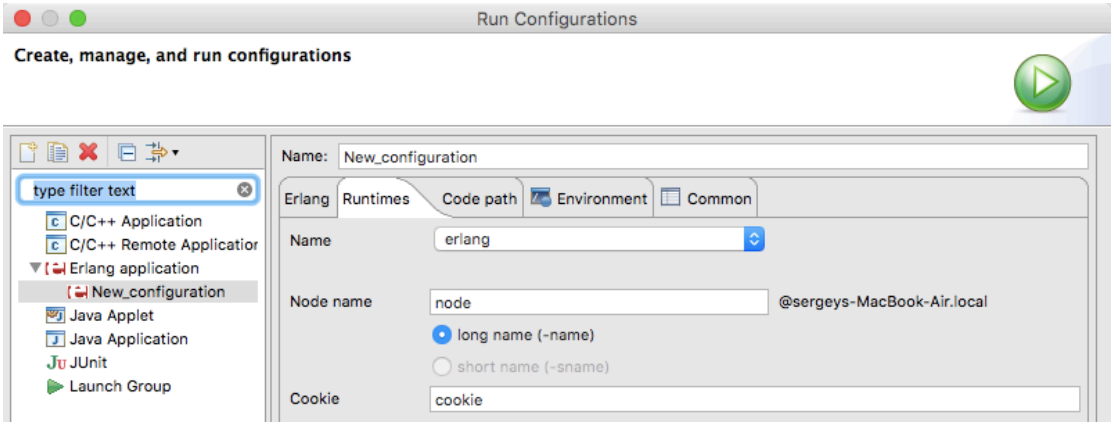
```
> erlang:get_cookie().
'RAWRQYREWZUNDKJSUSTI'
```

Функция set_cookie(Node, cookie) устанавливает известное значение cookie для узла.

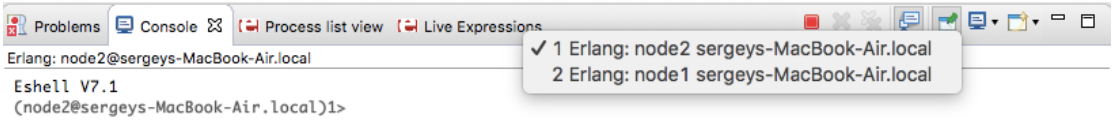
После изменения cookie одной из сторон другая сторона после перезапуска узла не сможет более запускать на нем процессы, при этом выдается сообщения о таких попытках:

```
(machine2@sergeys-Air.Dlink)3> erlang:set_cookie(node(),'secret').
true
(machine1@sergeys-Air.Dlink)1> erlang:get_cookie().
'RAWRQYREWZUNDKJSUSTI'
(machine1@sergeys-Air.Dlink)2> proc:main().
<0.42.0>
I'm ping, pong is : <0.48.0> <0.42.0>
 waiting for pong...
(machine1@sergeys-Air.Dlink)3>
=WARNING REPORT==== 12-Oct-2015::09:10:23 ===
** Can not start proc:pong,[] on 'machine2@sergeys-Air.Dlink' **
machine2@sergeys-Air.Dlink)4>
=ERROR REPORT==== 12-Oct-2015::09:10:23 ===
** Connection attempt from disallowed node 'machine1@sergeys-Air.Dlink' **
```

Чтобы запустить несколько узлов в среде разработки Eclipse можно создать несколько конфигураций запуска (Run->Run configurations) типа Erlang application, в каждой из которых задать разное имя узла и одинаковый cookie:




Далее после запуска у каждого узла будет своя консоль и консоли переключаются в среде:




При разработке приложений-серверов, работающих на удаленных узлах и принимающие сообщения Erlang можно поступать следующим образом – регистрировать pid сервера по глобальному псевдониму, который распространяется по узлам клиентов, и те могут отсылать сообщения, предварительно получив pid по этому псевдониму. Регистрация pid осуществляется при помощи global:register_name(name, pid), рассылка pid осуществляется по известным узлам из текущего узла (можно сделать ping узла, после чего автоматически распространятся все зарегистрированные имена).

- Сервер (node2):

```
runserver()->
      ServerPid = spawn(fun() -> worker() end),
global:register_name(server, ServerPid).

worker()->
      receive
      Msg-> io:format("Got msg=~p~n", [Msg])
      end,
      worker().
(node2@sergeys-Air.Dlink)1> test:runserver().
yes
```

```
(node2@sergeys-Air.Dlink)2> global:registered_names().
[server]
```

- Клиент (node1):

```
(node1@sergeys-Air.Dlink)1> net_adm:ping('node2@sergeys-Air.Dlink').
pong
(node1@sergeys-Air.Dlink)2> global:registered_names().
[server]
(node1@sergeys-Air.Dlink)3> server! hi.
** exception error: bad argument
     in operator  !/2
        called as server ! hi %прямо по атому имени посылать нельзя,
нужно вернуть pid по имени функцией whereis_name
(node1@sergeys-Air.Dlink)4> Pid=global:whereis_name(server).
<7160.70.0>
(node1@sergeys-Air.Dlink)5> Pid!hi.
hi
```

В результате сервер зарегистрировал имя server, клиент сделал ping до сервера и получил это имя, далее по нему получил pid удаленного сервера и может посылать ему сообщения.

**Задание к разделу 1.3**

1. Выбрать задачу реального мира из нескольких (5-6) взаимодействующих процессов
2. Промоделировать ее на языке UML (диаграмма Activity)
3. Реализовать ее работу через сообщения и вывод состояния
4. Запустить систему на нескольких работающих узлах
5. Установить возможность распараллеливания задания из раздела 1.2. Определить производительность (время решения) на больших исходных данных с использованием последовательного и параллельного подходов.