Язык Erlang

- Легкие процессы
- Параллельность, основанная на асинхронном обмене сообщениями
- Распределенность
- Отказоустойчивость
- Soft-Realtime
- Функциональный язык

Сферы применения

Распределенные, отказоустойчивые системы реального времени:

- Телекоммуникационные системы
- Интернет серверы (HTTP, Jabber, IMAP)
- Базы данных реального времени
- Другие применения.

Erlang не подходит для ”числодробилок”.

Используют Erlang

- Ericsson AXD301 ATM Switch
- Yaws – Yet Another Web Server
- Ejabberd – Erlang Jabber Daemon
- RabbitMQ
- И многое другое!

Простая задача

Есть множество сообщений, которые нужно отправить функцией doSend(msg).

Сообщение может не дойти, тогда нужно повторить отправку еще 4 раза, сделав задержку, соответственно 1, 2, 4 и 8 секунд.

Простая задача

- Сообщений немного, можно для каждого сообщения создать отдельный поток.

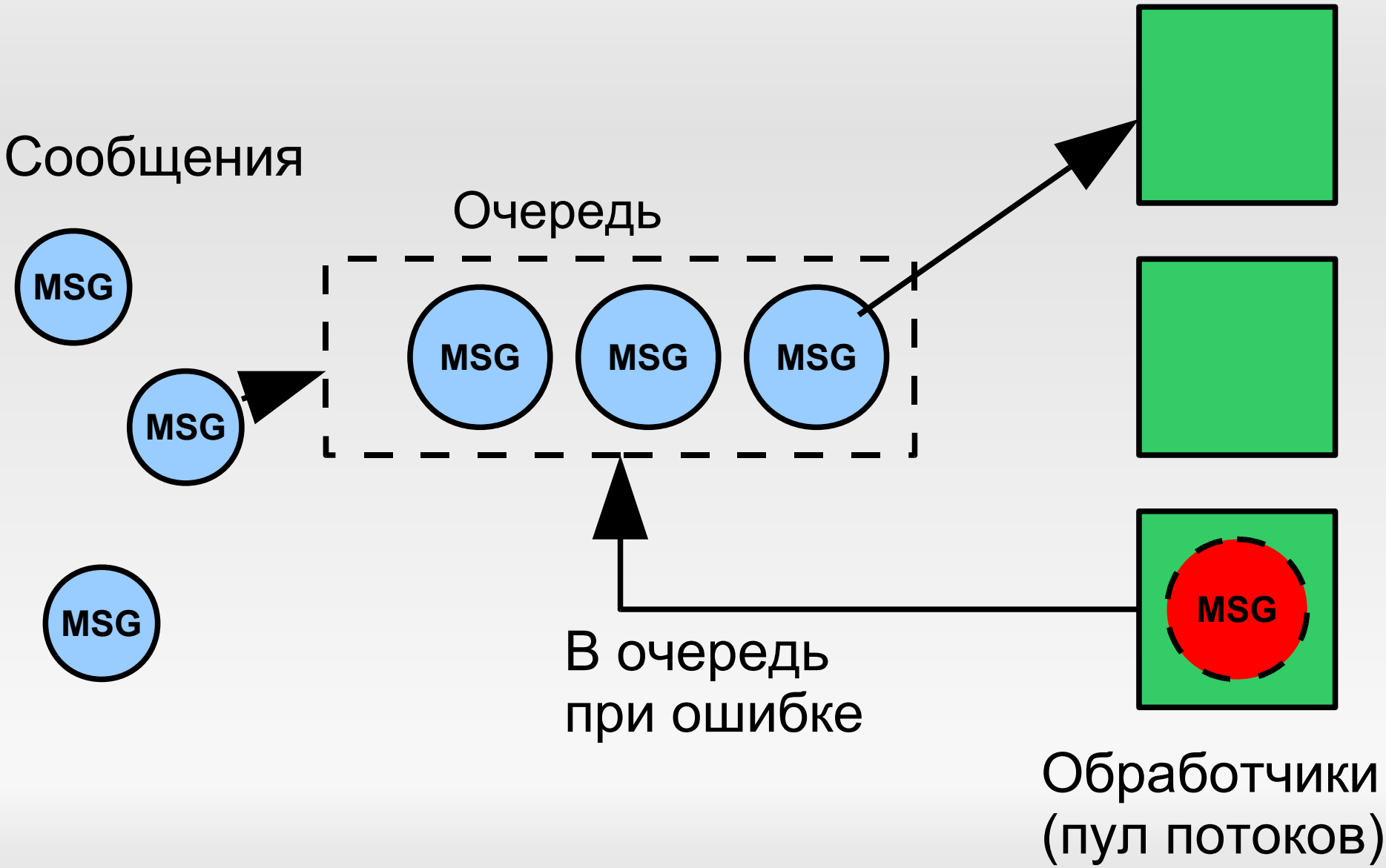
Простая логика

```
void send(Msg msg) { // В отдельном потоке
    boolean is_send = doSend(msg);
    int delay = 1;
    int n_try = 0;
    while (!is_send && n_try < 5) {
        Thread.sleep(delay*1000);
        is_send = doSend(msg);
        n_try++;
        delay *= 2;
    }
}
```

Усложним задачу

- Теперь одновременно отправляется множество сообщений (миллион), и возможности создать отдельный поток для каждого сообщения нет.

Сложная логика



Hello World

```
-module(hw).
-export([main/0]).

main() ->
    io:format(”Hello World”).
```

Синтаксис

- Нет циклов!
- Переменную можно сопоставить со значением только один раз
- Переменные начинаются с заглавной буквы
- Последнее выражение функции - возвращаемое функцией значение

Атомы

- Самоидентифицируемы
- Начинаются с маленькой буквы
- Могут заключатся в одинарные кавычки

Пример:

true

rectangle

'multi word atom'

Числа

- Целые
  1
  -999
  16#3F00
  2#1010101
  $Z
- Вещественные
  19.9293
  -303.13
  1.2E-20

Base#Value используется для задания числа в системе счисления Base

$Char используется для задания символа Char

Кортежи (Tuples)

- Контейнеры фиксированного размера
- Часто начинаются с атома:
  { ok, Socket }
  { error, Reason }
- Примеры
  { }
  { 123, { 123, { 123, [123] } } }
  { atom, 123, 16#FF, $A, ”Hello”, [1, 2, 3] }

Списки (Lists)

- Контейнеры переменной длины
- Конструкция [H | T] позволяет получить ”голову” и ”хвост” списка
  [H | T] = [1, 2, 3] % H = 1, T = [2, 3]
- Строка – это список.
  ”String” - список ASCII кодов символов, из который состоит слово ”String”.
- ”ABC” тоже самое, что [$A, $B, $C]

Binaries

- Низкоуровневая последовательность бит и байт
- Хорошо подходит для работы с бинарными протоколами
- Примеры

```
<<A:5, B:6>> = Bin % A – 5 bytes, B – 6 bytes
<<X:4/little-signed-integer-unit:8>> = Bin
Bin = <<10, A:4>>
```

Оператор ”=”

- **Не присваивание!**
- LHS = RHS
- Вычисляется RHS, затем переменные из LHS сопоставляются с конкретными значениями из RHS
  A = 5 % A == 5
  L = [1, 2, 3] % L = [1, 2, 3]

Pattern Matching

- A=10 (сопоставит A значение 10)
- {B, C, D} = {10, hello, vmk}
  % B=10, C=hello, D=vmk
- {A, A, B} = { one, one, two}
  % A=one, B=two
- {A, A, B} = { one, two, three }
  Ошибка

Pattern Matching #2

- `[A,B|C]=[1,2,3,4]`
  `% A=1,B=2, C=[3,4]`
- `[A|B]=[hello]`
  `% A=hello, B=[]`
- `[A|B]=[]` Ошибка

Guards

- Простые условия, дополняющие Pattern Matching
- Делают код понятнее и проще
- Пример функции:

```
max(A, B) when A > B -> A;
max(A, B) -> B.
```

- Операции сравнения: <, <=, ...
- Проверка типа: is_list(A), ...

Объявление функции

```
func(A, B, ...) where A > B ->
    ... ;

func(Pattern1, Pattern2, ...) ->
    ... ;

    ...

func(Pattern1, Pattern2, ...) ->
    ... .
```

If и Case

```
case Expr of
Pattern1 [when Guard] ->
        Body1;
    ...;
PatternN [when Guard] ->
        BodyN
end
```

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
end
```

Анонимные функции

- Создание анонимной функции:

```
F = fun(A) -> 2*A end
```

- Пример использования

```
F = fun(A) -> 2*A end,
List = map(F, [1, 2, 3, 4])
% List == [2, 4, 6, 8]
```

- Передаются как параметры в функции высших порядков

Функция foreach

Применяет F(A) ко всем элементам списка [A1, A2, ...]:

```
foreach(F, [Hd|Tail]) ->
    F(Hd),
    foreach(F, Tail);
foreach(F, []) when
 is_function(F, 1) -> ok.
```

Функция map

Применяет F(A) ко всем элементам списка [A1, A2, ...] и возвращает список результатов [F(A1), F(A2), ...]:

```
map(F, [H|T]) ->
    [F(H)|map(F, T)];
map(F, []) when
 is_function(F, 1) -> [].
```

Tail-Recursive

```
loop(A) ->
     B = A + 1,
     C = B – 1,
     loop(C).
```

```
int loop(int a) {
   int b = a + 1;
   int c = b – 1;
   return loop(c);
}
```

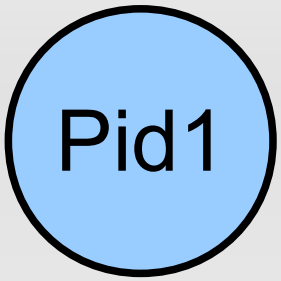
Last Call Optimization

```
g(X) -> ..., h(X).
h(X) -> ..., i(X).
i(X) -> ..., g(X).
```

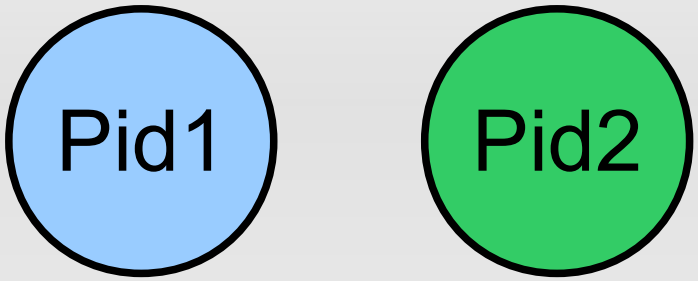
Функция g(X) будет выполняться в огранниченой области памяти.

Создание процесса

До:



После:

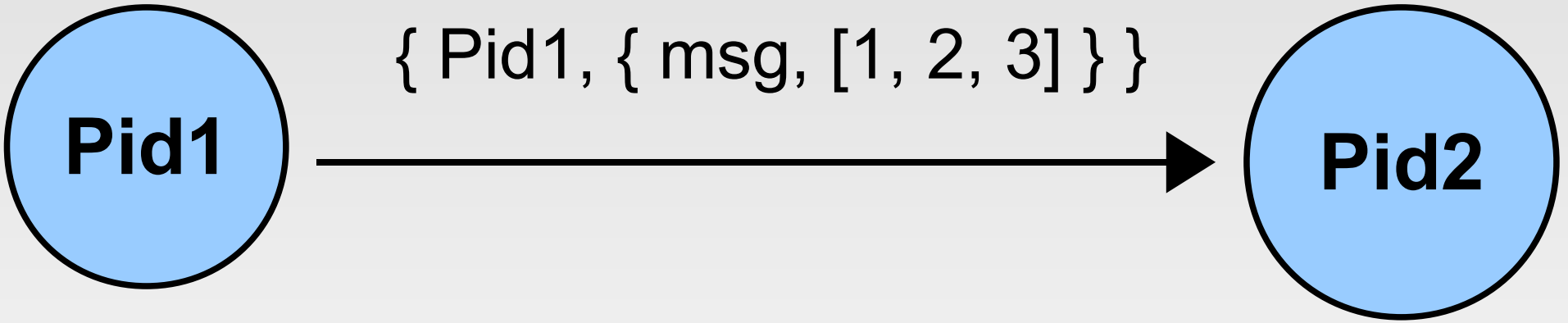


Код из процесса Pid1, создающий Pid2:

```
…
Pid2 = spawn(Mod, Func, Args)
…
```

Pid2 – идентификатор созданного процесса.
Pid2 известен процессу Pid1.

Отправка сообщений




```
Pid2 ! { self(),
    { msg, [1, 2, 3] }
}
```

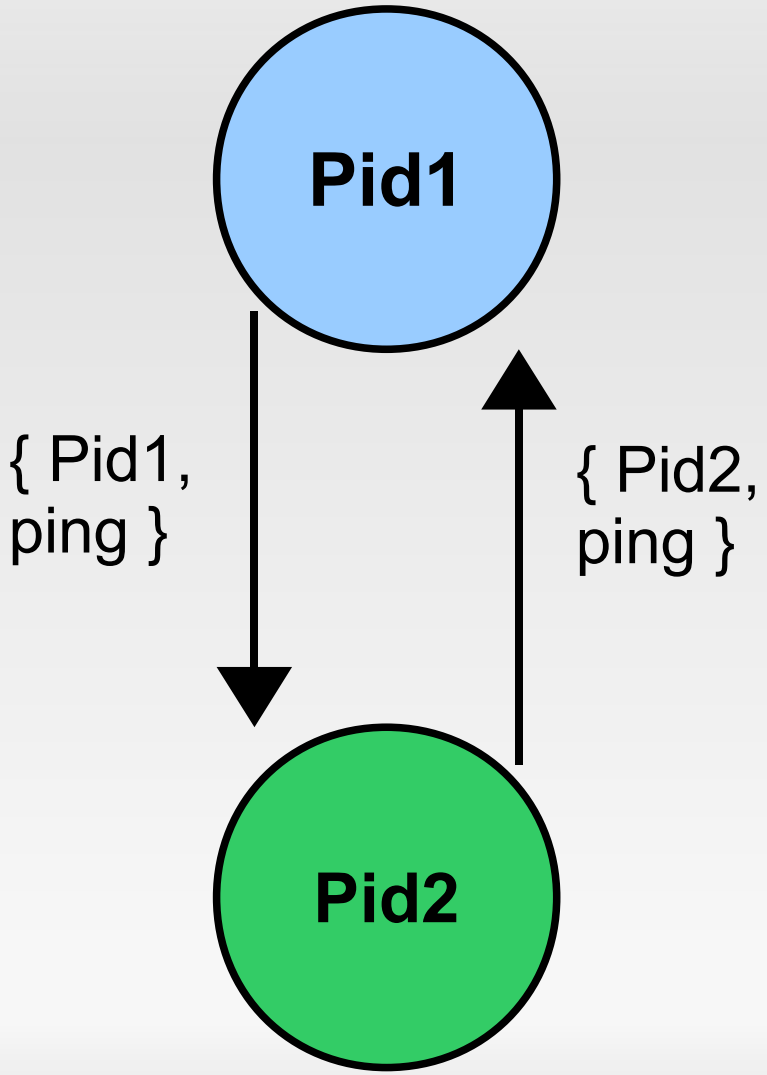
```
receive
    { Pid, { msg, Data } } ->
        process(Data)
end.
```

Эхо-процесс

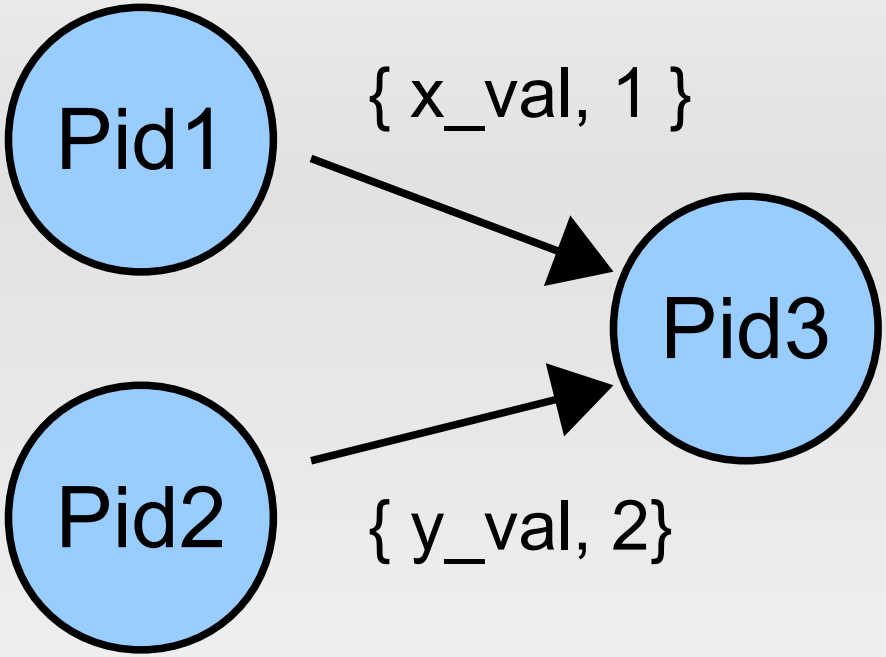
```
-module(echo).
-export([init/0, loop/0]).

init() ->
    Pid2 = spawn(echo, loop, []),
    Pid2 ! { self(), ping },
    receive
        { Pid2, Msg } ->
            io:format("~w~n", [Msg])
    end.

loop() ->
    receive
        { From, Msg } ->
            From ! { self(), Msg },
            loop()
    end.
```



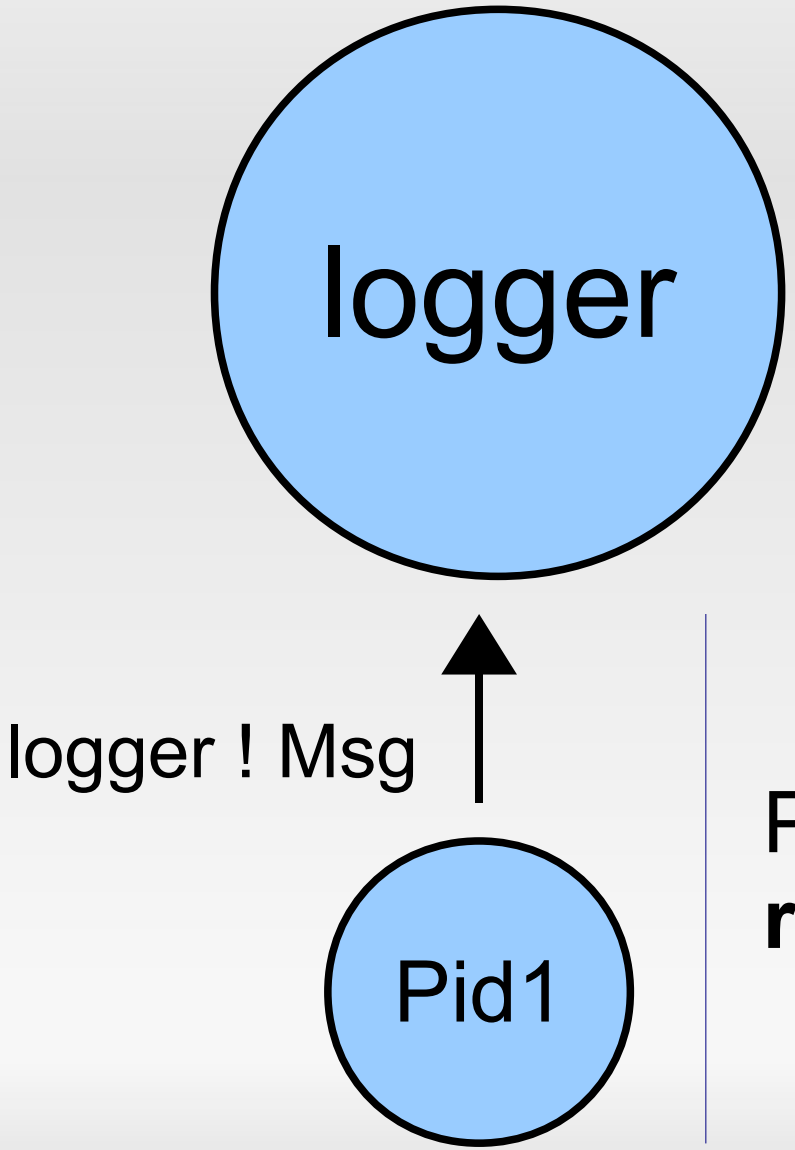
Выборочный прием сообщений




```
receive
    { x_val, X } ->
        use(X)
end,
receive
    {y_val, Y } ->
        use(Y)
end.
```

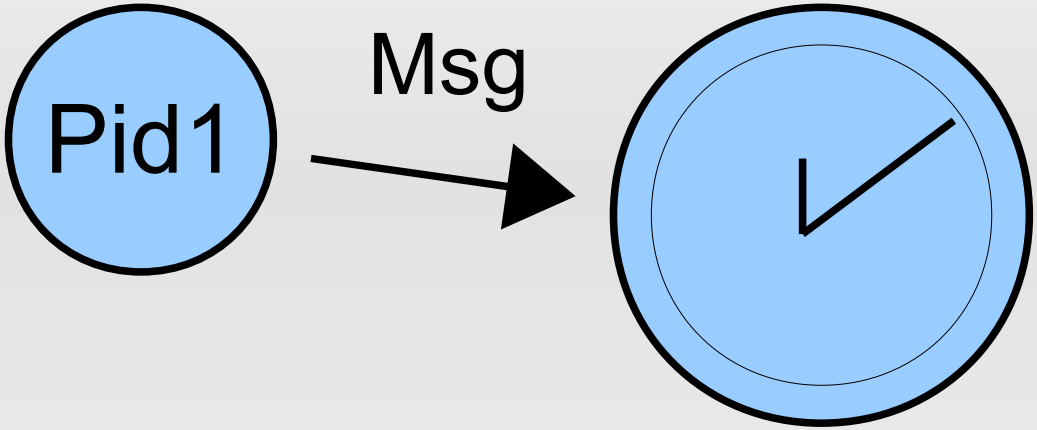
Сообщения процессу **Pid3** могут прийти в произвольном порядке.
Сообщение { x_val, 1 } будет обработано первым, затем будет обработано { y_val, 2 }.

Именованный процесс



Pid2 = spawn(Mod, Fun, Arg),
register(logger, Pid2)

Таймауты

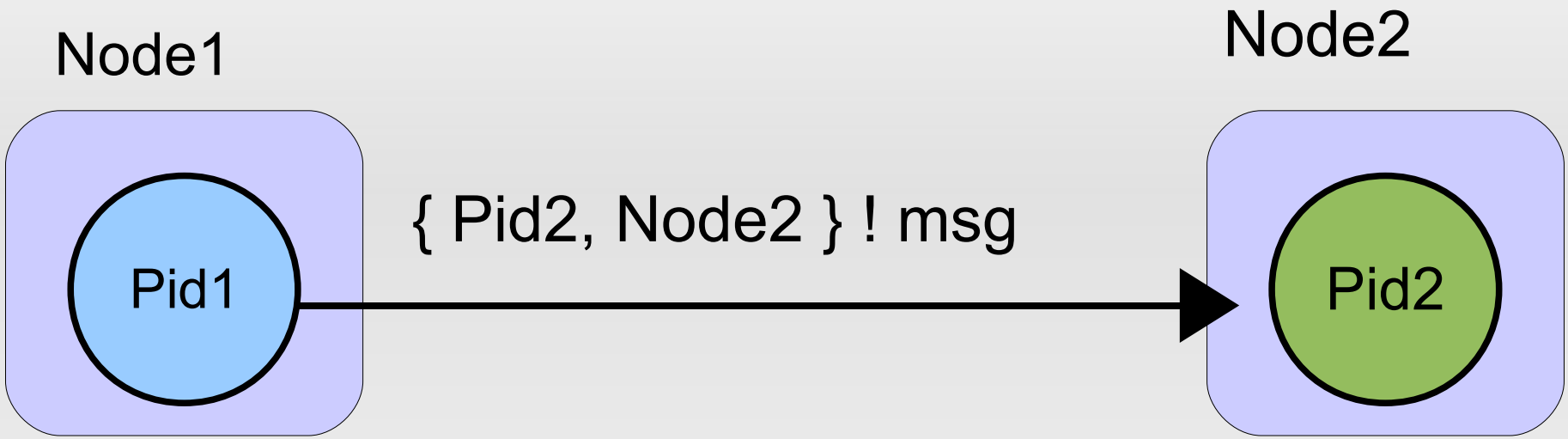



```
receive
    Msg ->
        process(Msg)
    after 1000 ->
        io:format(”Timeout”)
end.
```

Легкие процессы

- Процессы стартуют с маленькой кучей (309 слов). При необходимости размер увеличивается.
- Эффективнее используется кэш
- Нет блокировок (нет общих данных)

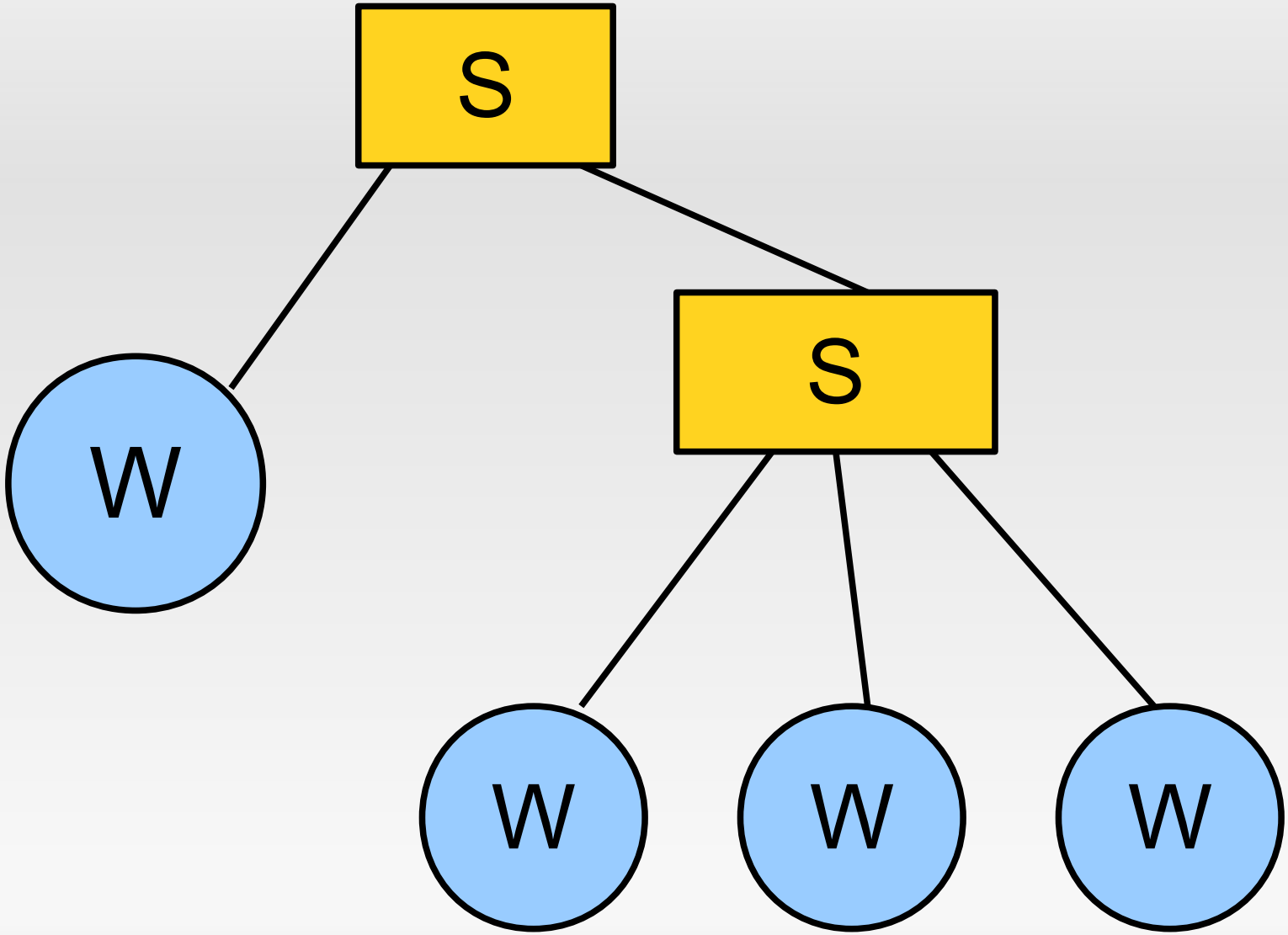
Распределенность



Pid1 создает процесс Pid2 на Node2:

```
% создаем процесс
Pid2 = spawn(Node2, Mod, Fun, Args),
% отправляем сообщение
{ Pid2, Node2 } ! msg
```

Supervision Tree



Workers & Supervisors

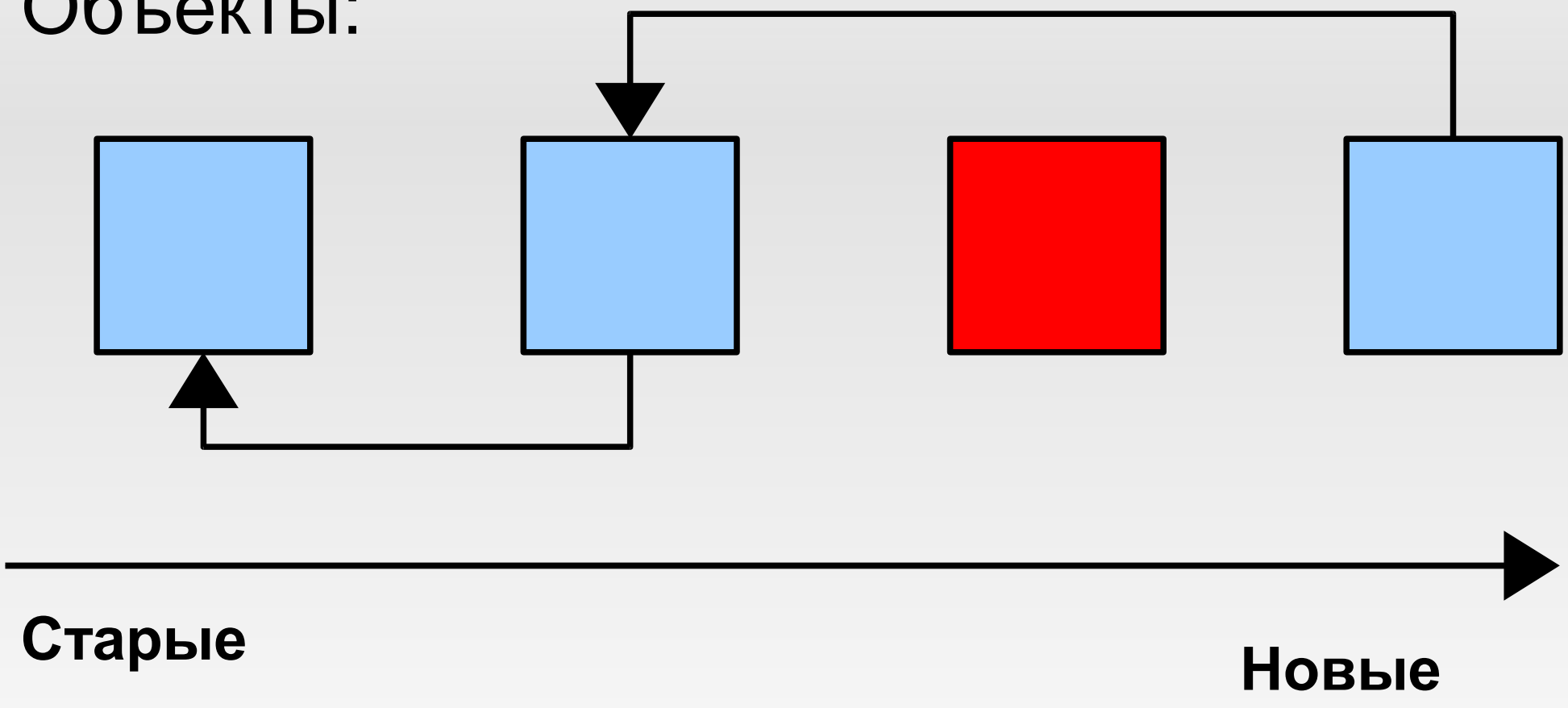
- Workers – процессы, которые непосредственно выполняют всю работу
- Supervisors – процессы, которые следят за Worker'ами и при необходимости перезапускают их
- Стратегии перезапуска: **one_for_one**, **one_for_all**, **rest_for_one**.

Soft Realtime

- Hard Realtime – гарантия: ”время отклика всегда будет менее 50 миллисекунд”.
- Гарантия: ”В 97% случаев время отклика системы составит менее 50 миллисекунд” - это Soft Realtime.
- Erlang:
  1) Нет Lazy Evaluation
  2) Сборка мусора производится отдельно для каждого процесса
  3) Быстрое переключение между процессами

Garbage Collector

Объекты:



Интересные темы

- Erlang/OTP: app, gen_server, gen_fsm, ...
- Обработка ошибок
- Горячая замена кода
- Взаимодействие с C и Java
- Embedded использование

Полезные ссылки

- http://erlang.org
- http://trapexit.org/
- http://spawnlink.com
  Перевод: http://tiesto.habrahabr.ru/blog/

Спасибо за внимание!

- E-mail: romkhat@gmail.com
- Jabber: nosorog@jabber.ru