

Programowanie współbieżne

Programowanie w języku Erlang

- Model aktorów
- Współbieżność w Erlangu
- Tworzenie procesów
- Komunikacja
- Rejestr procesów
- Wzorce projektowe
- Dynamiczne ładowanie kodu
- Programowanie rozproszone

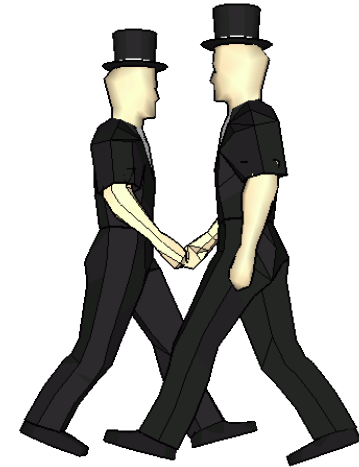
Model aktorów

- 1973, Carl Hewitt
- Jawne zdefiniowanie paradygmatu dla współbieżności
- Program składa się z aktorów
- Aktor jest niezależnym potokiem przetwarzania instrukcji
 - Nie synchronizuje wykonania z innymi aktorami
- Aktor może:
 - Utworzyć innych aktorów
 - Wysyłać komunikaty do innych aktorów
 - Odbierać komunikaty od innych aktorów i reagować na nie



Komunikacja aktorów

- Wysłanie komunikatu wymaga adresu odbiorcy
- Aktor zna własny adres
- Aktor zna adres aktora, którego utworzył
- Adres można przekazać w komunikacie



- Kolejność przekazywania komunikatów nie jest ustalona
- Komunikaty do przetworzenia mogą być buforowane, ale bufor jest skończony
- Komunikaty nie znikają
- Nadawca nie jest informowany o dostarczeniu komunikatu

Współbieżność w Erlangu

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne
- Procesy nie dzielą pamięci

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne
- Procesy nie dzielą pamięci
- Procesy są sekwencyjne

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne
- Procesy nie dzielą pamięci
- Procesy są sekwencyjne
- Procesy są asynchroniczne

Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne
- Procesy nie dzielą pamięci
- Procesy są sekwencyjne
- Procesy są asynchroniczne
- Procesy mogą tworzyć inne procesy

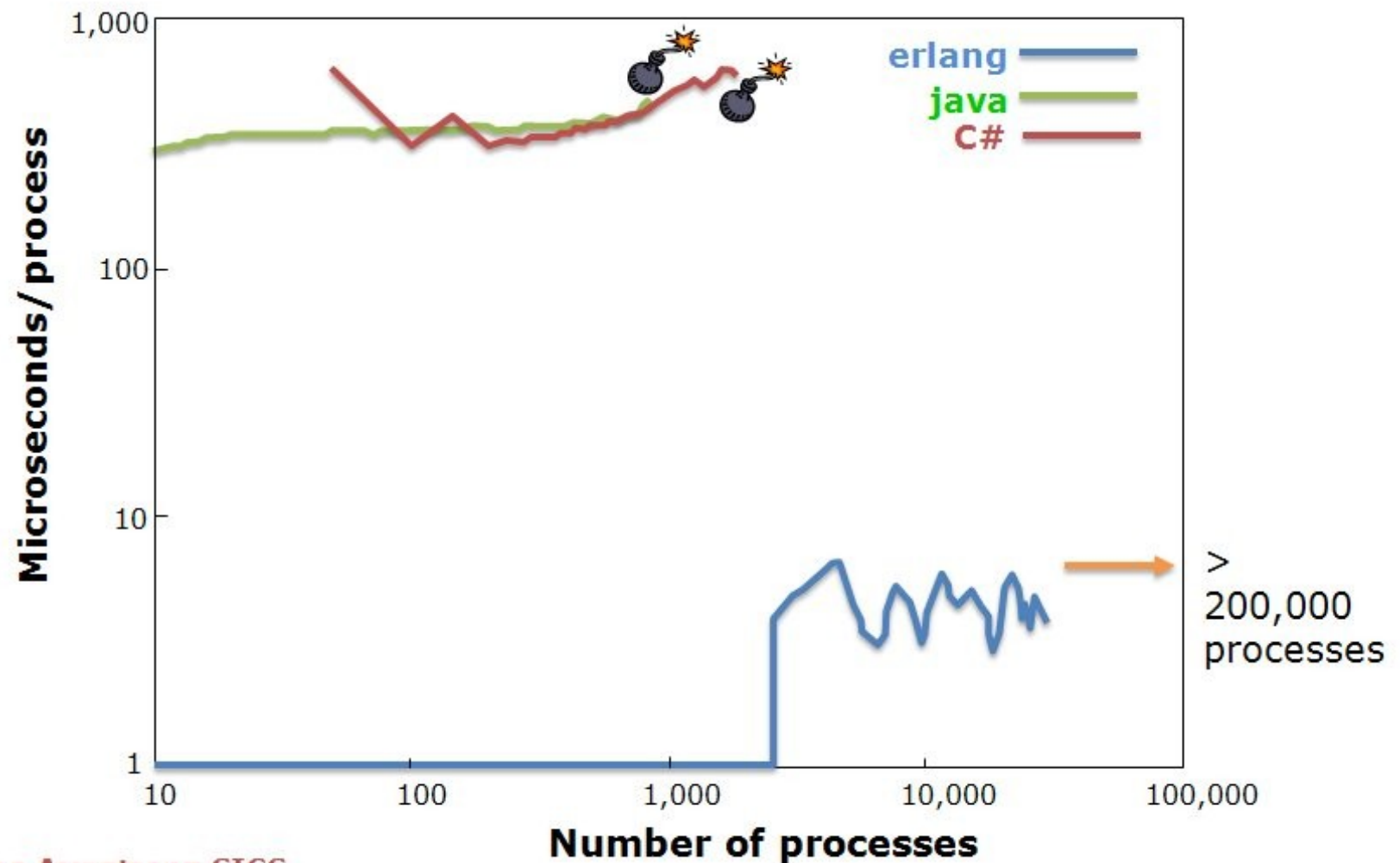
Założenia modelu współbieżności

- Maszyna wirtualna Erlanga wykonuje kod programów w **procesach**
- Procesy są równoprawne
- Procesy nie dzielą pamięci
- Procesy są sekwencyjne
- Procesy są asynchroniczne
- Procesy mogą tworzyć inne procesy
- Procesy komunikują się za pomocą asynchronicznych komunikatów

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Tworzenie procesów

Process creation times



Source: Joe Armstrong SICs

Tworzenie procesu

- BIF *spawn* tworzy nowy proces
- Zwraca identyfikator nowego procesu (*PID*)
- Nowy proces zaczyna działanie od wykonania funkcji *function* z modułu *module* z argumentami z listy *arguments* - (MFA)
- BIF *spawn* zawsze kończy się natychmiast i nigdy nie zawodzi

```
Pid = spawn(module, function, arguments)
```

```
Pid2 = spawn(fun)
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Komunikacja

Wysyłanie wiadomości

- *PID* zwrócony przez funkcję *spawn* jest adresem, który jest niezbędny do wysłania wiadomości
- Do wysyłania wiadomości służy operator **!**
- Wysłać można dowolną poprawną strukturę danych
- Wysyłanie wiadomości zawsze kończy się natychmiast i nigdy nie zawodzi
- Wiadomości wysłane do nieistniejących procesów są ignorowane

```
Pid = spawn(module, function, arguments)
Pid ! hello
```

- Odbierane wiadomości są zapisywane w skrzynce odbiorczej procesu
- Odbieranie jest realizowane wyrażeniem *receive*
- Wiadomość jest dopasowywana do wzorców

```
receive
  hello -> io:format("Hello World!");
  bye   -> io:format("Bye World!");
        -> unknownMessage
end.
```

Odbieranie wiadomości

- Odbierane wiadomości są zapisywane w skrzynce odbiorczej procesu
- Odbieranie jest realizowane wyrażeniem *receive*
- Wiadomość jest dopasowywana do wzorców

```
receive
  hello -> io:format("Hello World!");
  bye   -> io:format("Bye World!");
        -> unknownMessage
end.
```

Kolejka wiadomości

- Wyrażenie *receive* wstrzymuje wykonanie procesu do czasu dopasowania którejś z wiadomości do wzorca
- Wiadomości są dopasowywane do wzorców poczynając od najstarszej
- Jeśli wiadomość nie pasuje do żadnego wzorca, jest zwracana do kolejki

Wzorzec przekazywania PIDu nadawcy

- Jeśli proces odbierający wiadomość ma na nią odpowiedzieć, musi znać PID nadawcy
- Do pobrania własnego PIDu służy BIF *self()*

```
-module(mod) .  
-export([createAndAsk/0, reply/0]).  
  
createAndAsk() ->  
    Pid = spawn(mod, reply, []),  
    Pid ! {self(), question},  
    receive  
        answer -> io:format("Received!")  
    end.  
  
reply() ->  
    receive  
        {Pid, question} -> Pid ! answer  
    end.
```

Procesy są lekkie

```
-module(demo_03_processCreation).  
  
-export([pass/1, pass/0]).  
  
pass(Count) ->  
    Pid = spawn(?MODULE, pass, []),  
    Pid ! {self(),Count},  
    receive  
        _ -> ok  
    end.  
  
pass() ->  
    receive  
        {ParentPid, Count} when Count > 0 ->  
            Pid = spawn(?MODULE, pass, []),  
            Pid ! {self(),Count - 1},  
            receive  
                _ -> ParentPid ! ok  
            end;  
        {ParentPid,_} -> ParentPid ! ok  
    end.
```


Wzorzec odbierania selektywnego

- Gdy kolejność przetwarzania wiadomości może być dowolna:

```
receive
  Msg -> processMessage (Msg)
end.
```

- Gdy kolejność przetwarzania wiadomości ma znaczenie:

```
receive
  {msg1, Data} -> processMessage1 (Data)
end,
receive
  {msg2, Data} -> processMessage2 (Data)
end.
```

Czas oczekiwania na wiadomość

- Wyrażenie *receive* wstrzymuje wykonanie procesu do czasu dopasowania otrzymanej wiadomości
- Opcjonalnie może zakończyć się bez dopasowania po określonym czasie

```
receive
  msg1 ->
    processMsg1 ();
  msg2 ->
    processMsg2 ()
after
  1000 ->
    cancelProcessing()
end.
```

Wzorzec opróżniania skrzynki wiadomości

- Wiadomości otrzymane są gromadzone w skrzynce odbiorczej
- Gromadzenie nieprzetwarzanych wiadomości może przepełnić pamięć

```
flush() ->  
  receive  
    -> flush()  
  after  
    0 -> ok  
end.
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Rejestr procesów

Rejestrowanie procesów

- *PID* jest znany tylko twórcy procesu
- *PID* można przesyłać do innych procesów, ale...
- Proces można zarejestrować pod nazwą (aliasem)
- Każdy proces może wysłać wiadomość do procesu zarejestrowanego pod nazwą

```
register(alias, Pid),  
alias ! Msg.
```

- Wysyłanie wiadomości z wykorzystaniem aliasu do niezarejestrowanego procesu kończy się błędem
- BIF *registered()* zwraca wszystkie zarejestrowane aliasy
- BIF *whereis(alias)* zwraca PID procesu zarejestrowanego

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Wzorce projektowe

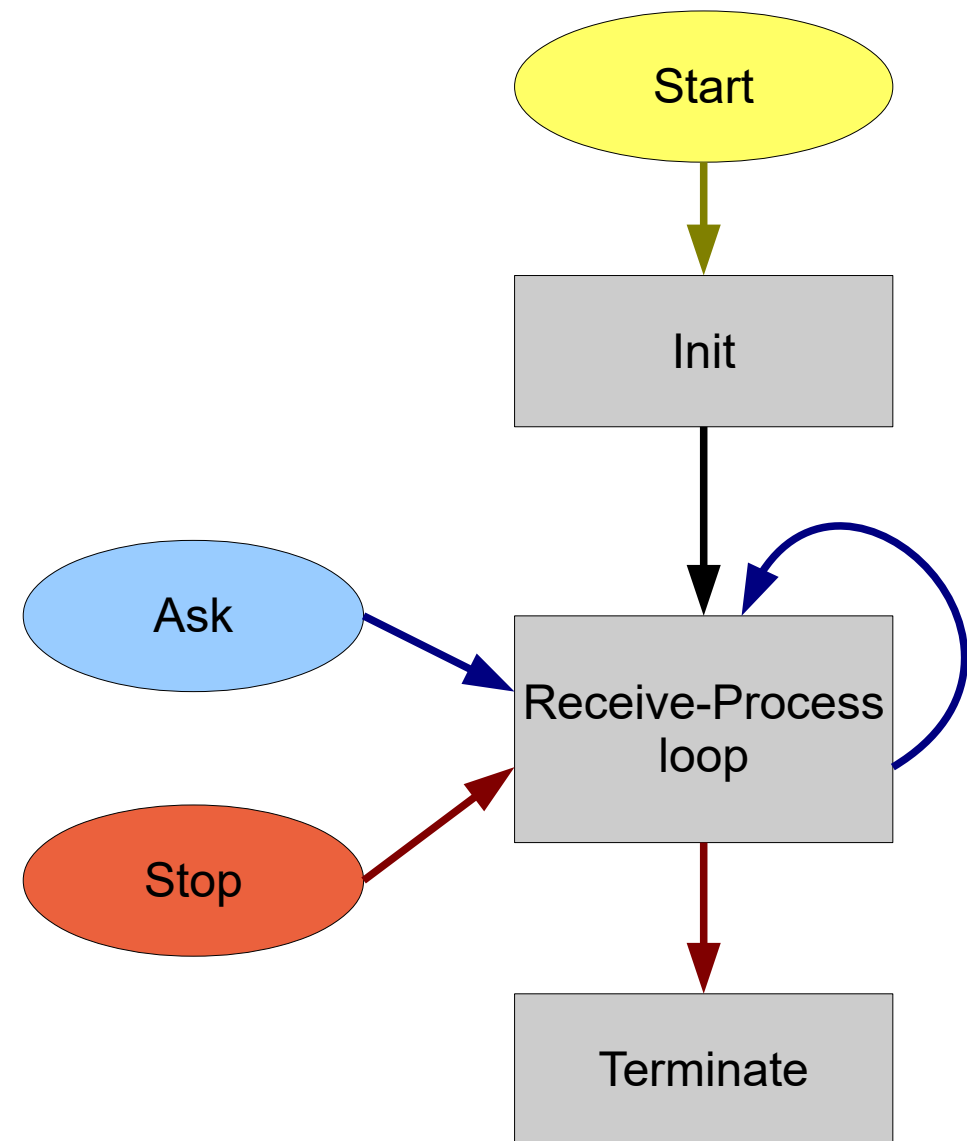
Ogólny wzorzec usługi

```
start(Args) ->
  spawn(server, init, Args).

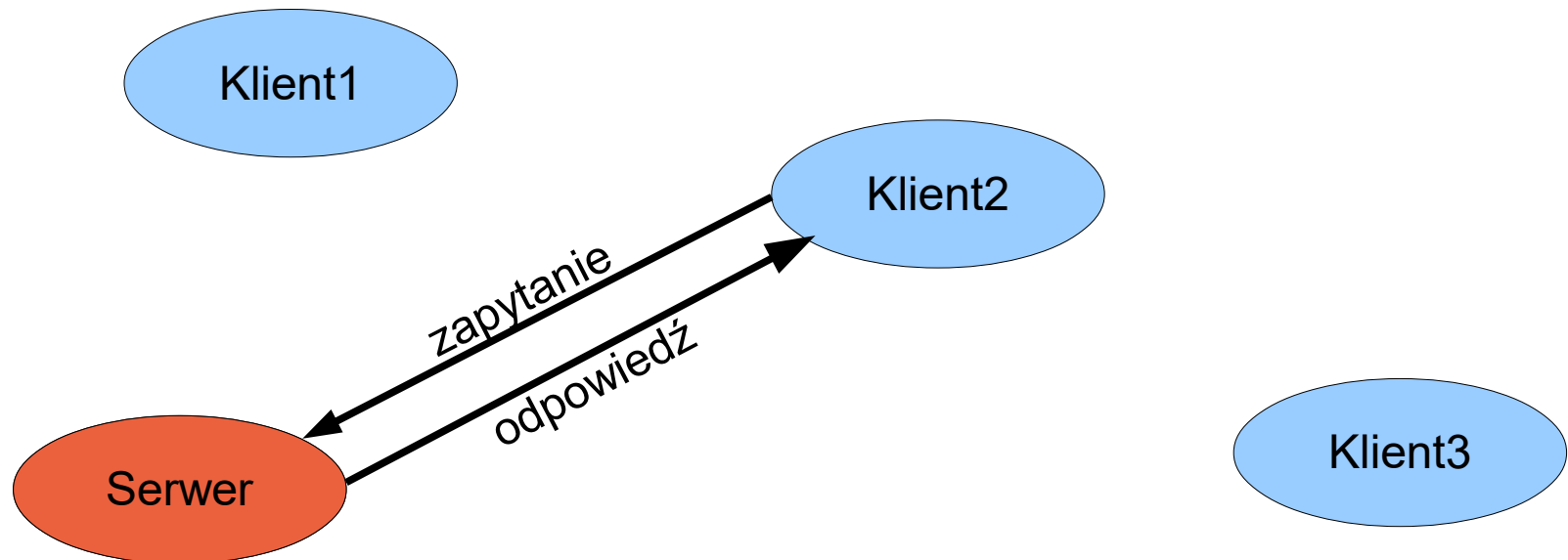
init(Args) ->
  State = initState(Args),
  loop(State).

loop(State) ->
  receive
    {msg, Msg} ->
      NewState = handle(Msg),
      loop(NewState);
    stop ->
      terminate()
  end.

terminate() ->
  ok.
```



- Zasób współdzielony przez wielu klientów
- Odpowiada na zapytania w sposób synchroniczny



Serwer mnożący przez 3

- Serwer będzie mnożył przesłaną liczbę przez 3 i odsyłał wynik do nadawcy.

```
-module(mulServer).  
  
-export([start/0, stop/0, mul/1]).  
-export([loop/0]).  
  
start() ->  
    register (m3server, spawn (?MODULE, loop, [])).  
  
loop() ->  
    receive  
        stop -> ok;  
        {Pid, N} ->  
            Result = N * 3,  
            io:format("Server got ~w, returning ~w ~n", [N, Result]),  
            Pid ! Result,  
            mulServer:loop()  
    end.
```

Serwer mnożący przez 3

- API

```
stop() ->
    m3server ! stop.

mul(N) ->
    m3server ! {self(), N},
    receive
        M -> M
    end,
    io:format("Result: ~p~n", [M]).
```

Serwer zmiennej globalnej

- Serwer będzie przechowywał wartość całkowitoliczbową, którą klient będzie mógł zwiększyć, zmniejszyć lub pobrać

```
-module(var) .  
  
-export([start/0, stop/0, inc/0, dec/0, get/0]).  
-export([init/0]).  
  
start() ->  
    register (varServer, spawn(var, init, [])).  
  
init() ->  
    loop(0).
```

Serwer zmiennej globalnej

```
%% main server loop

loop(Value) ->
  receive
    {request, Pid, inc} ->
      Pid ! {reply, ok},
      loop(Value + 1);

    {request, Pid, dec} ->
      Pid ! {reply, ok},
      loop(Value - 1);

    {request, Pid, get} ->
      Pid ! {reply, Value},
      loop(Value);

    {request, Pid, stop} ->
      Pid ! {reply, ok}
  end.
```

- Ukrywanie implementacji protokołu

```
%% client

call(Message) ->
    varServer ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

inc() -> call(inc).
dec() -> call(dec).
get() -> call(get).
stop() -> call(stop).
```

Dynamiczne ładowanie kodu

Przykład: serwer mnożący raz jeszcze

```
start() ->
    register (m3server, spawn (?MODULE, loop, [])).

loop() ->
    receive
        stop -> ok;
        {Pid, N} ->
            Result = N * 6,
            io:format("Server got ~w, returning ~w ~n", [N, Result]),
            Pid ! Result,
            mulServer:loop()
    end.

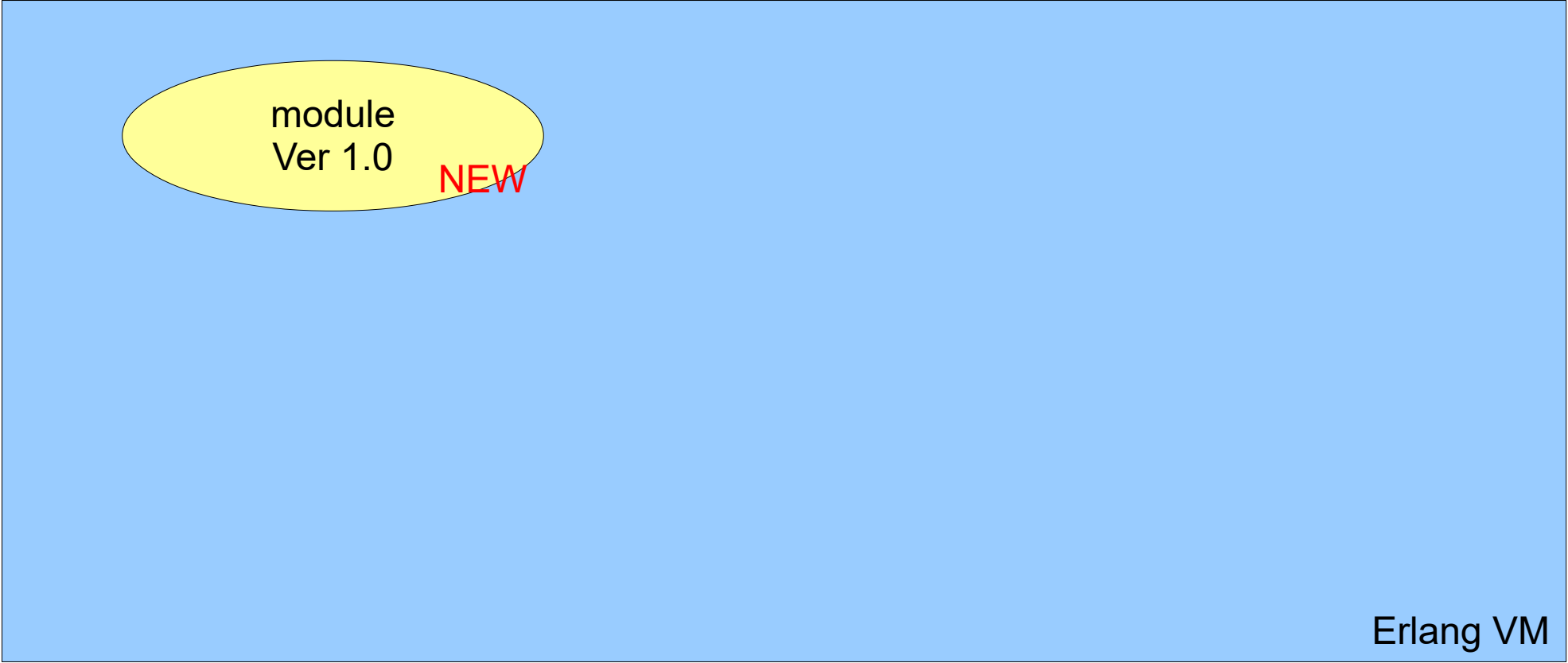
stop() ->
    m3server ! stop.

mul(N) ->
    m3server ! {self(), N},
    receive
        M -> M
    end,
    io:format("Result: ~p~n", [M]).
```


- Erlang przechowuje dwie wersje każdego modułu
- Każdy proces może wykonywać dowolną z nich
- W chwili wywołania funkcji z podaniem nazwy modułu proces przełącza się na najnowszą wersję kodu modułu

- Narzędzie zajmujące się ładowaniem modułów
- Utrzymuje dwie wersje każdego modułu: **nową** i **starą**
- Pozwala na ładowanie nowych wersji modułów - zmienia wtedy aktualną **nową** w **starą**, a poprzednią **starą** usuwa
- Likwiduje procesy, które korzystają z aktualnej **starej** wersji w chwili pojawienia się nowej **nowej**

Wersje modułów

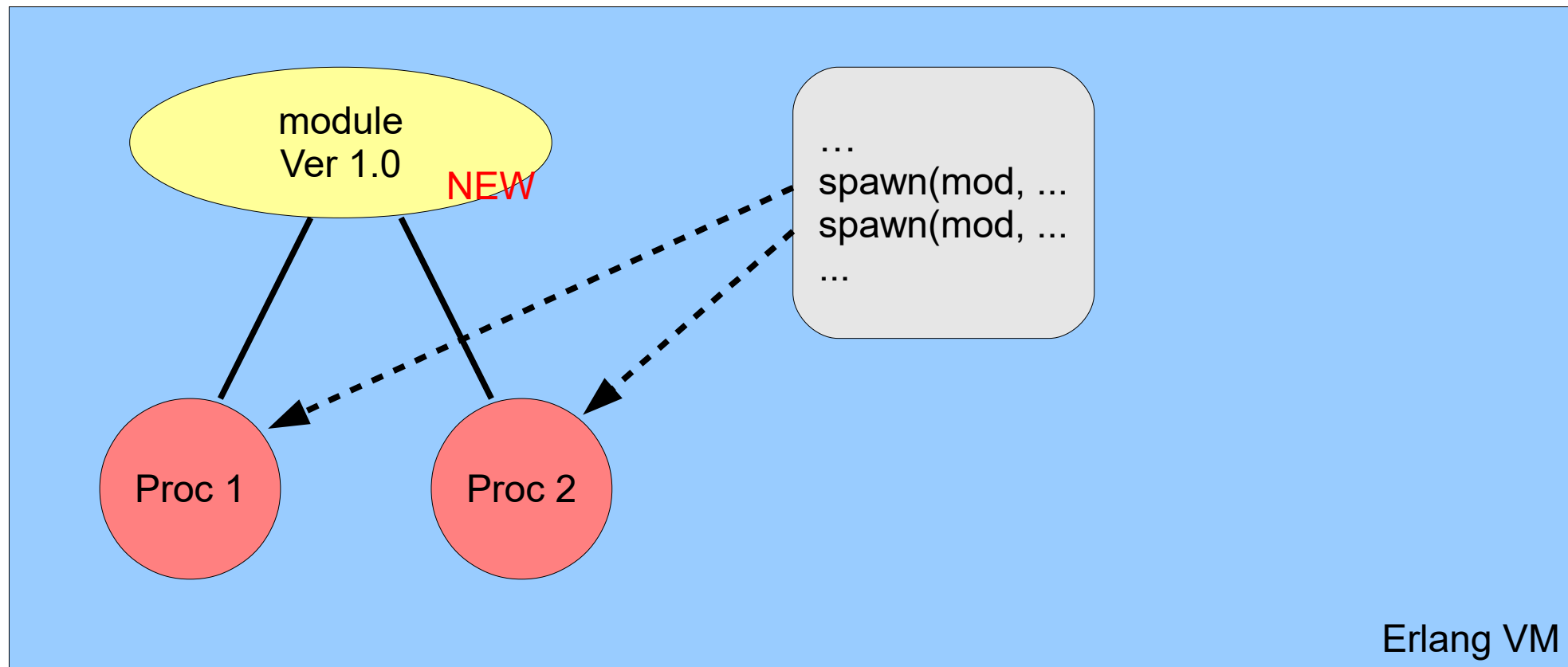


The diagram shows a large light blue rectangle representing the Erlang VM. Inside the top-left corner of this rectangle is a yellow oval. Inside the yellow oval, the text "module Ver 1.0" is written in black, and the word "NEW" is written in red to its right.

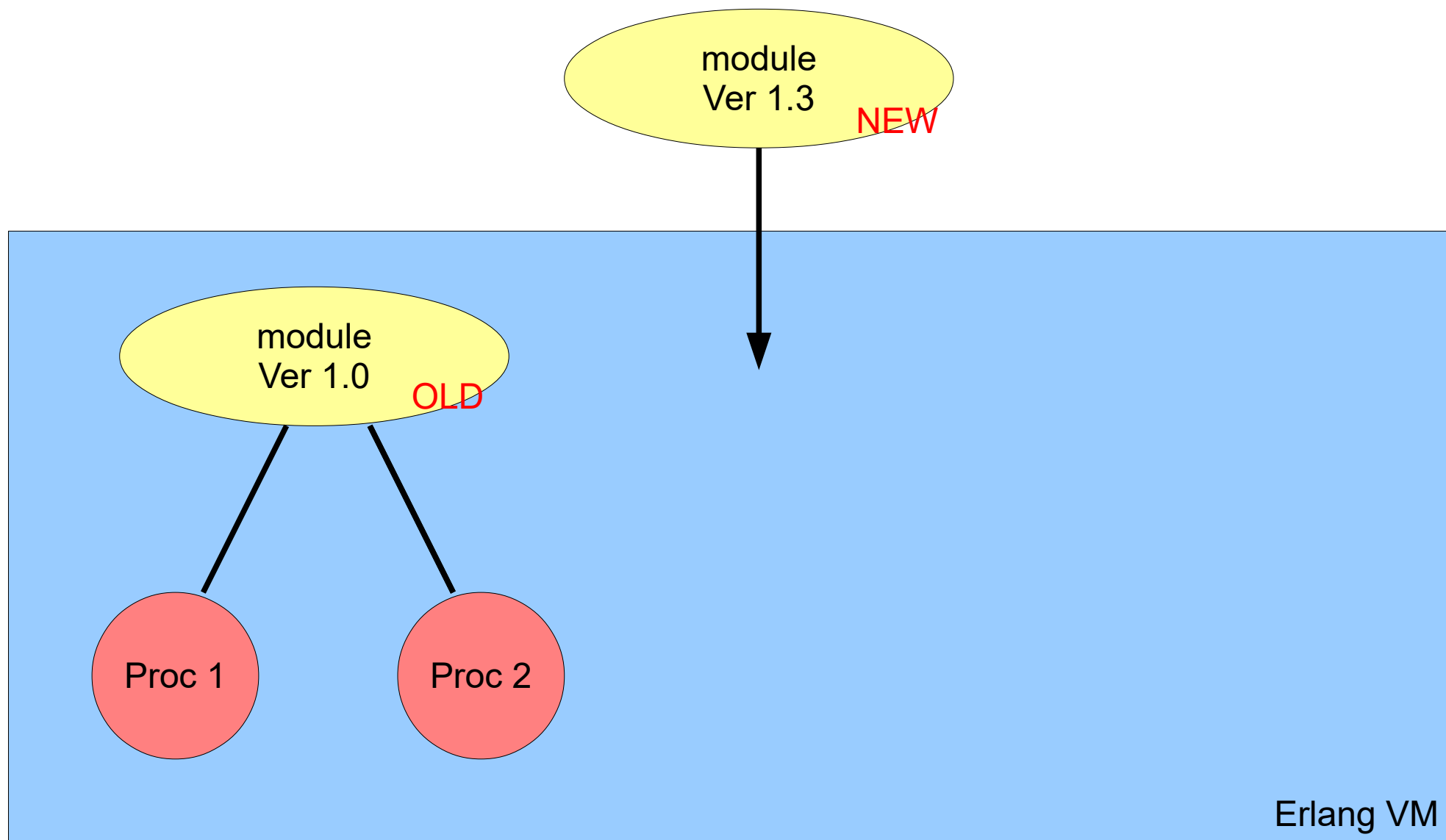
module
Ver 1.0 NEW

Erlang VM

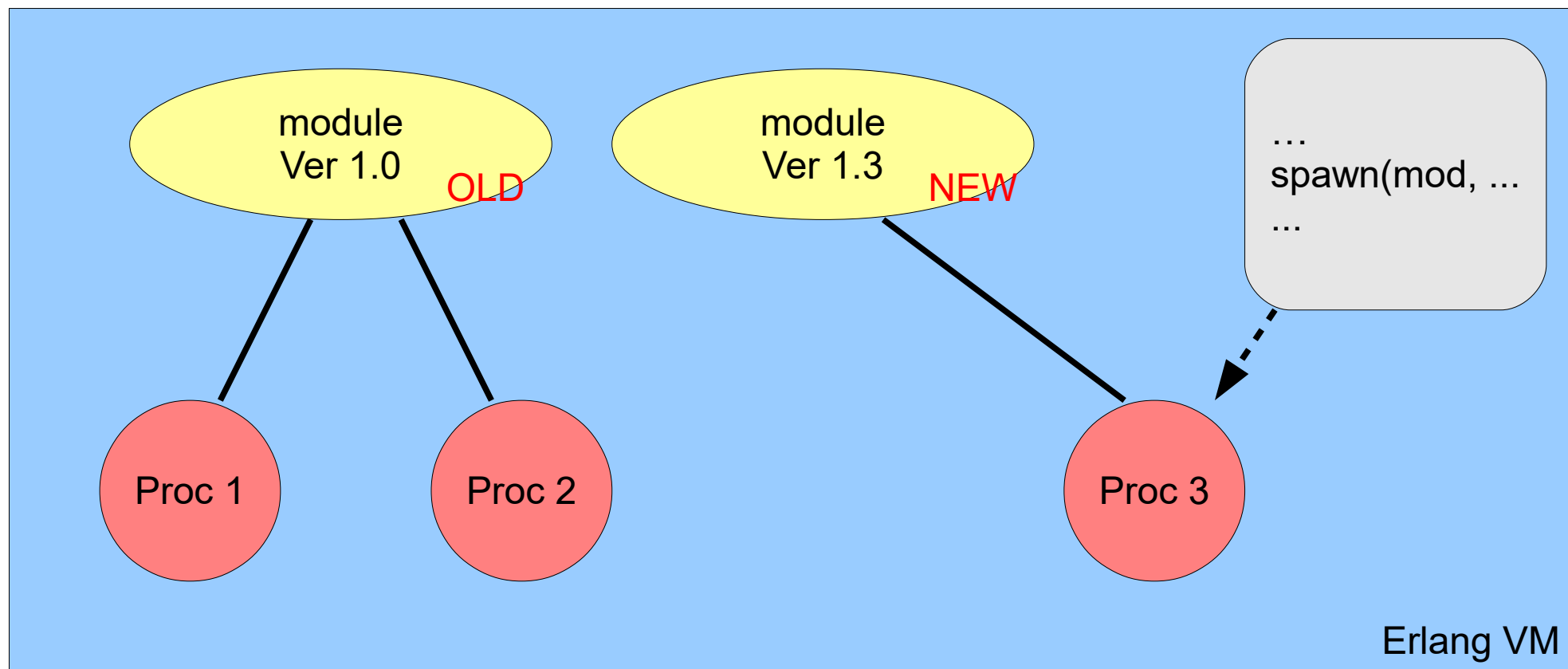
Wersje modułów



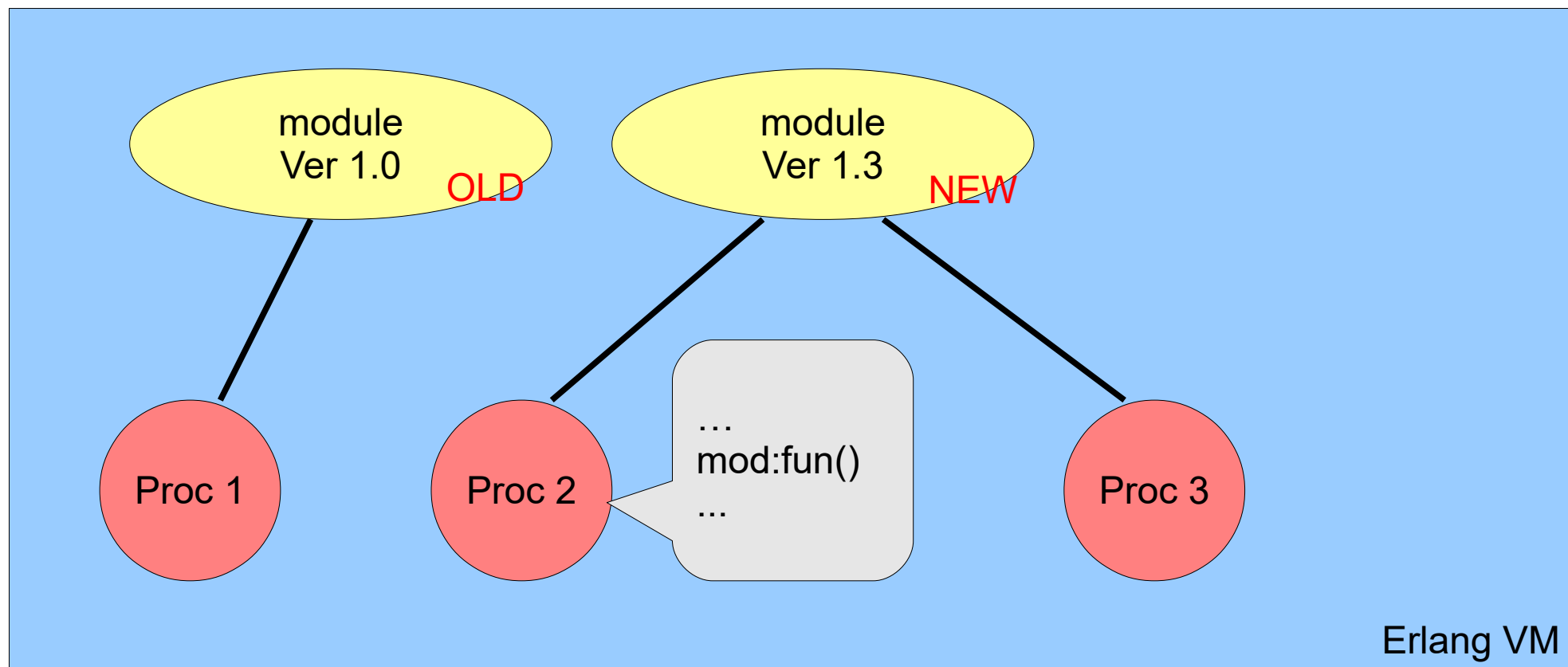
Wersje modułów



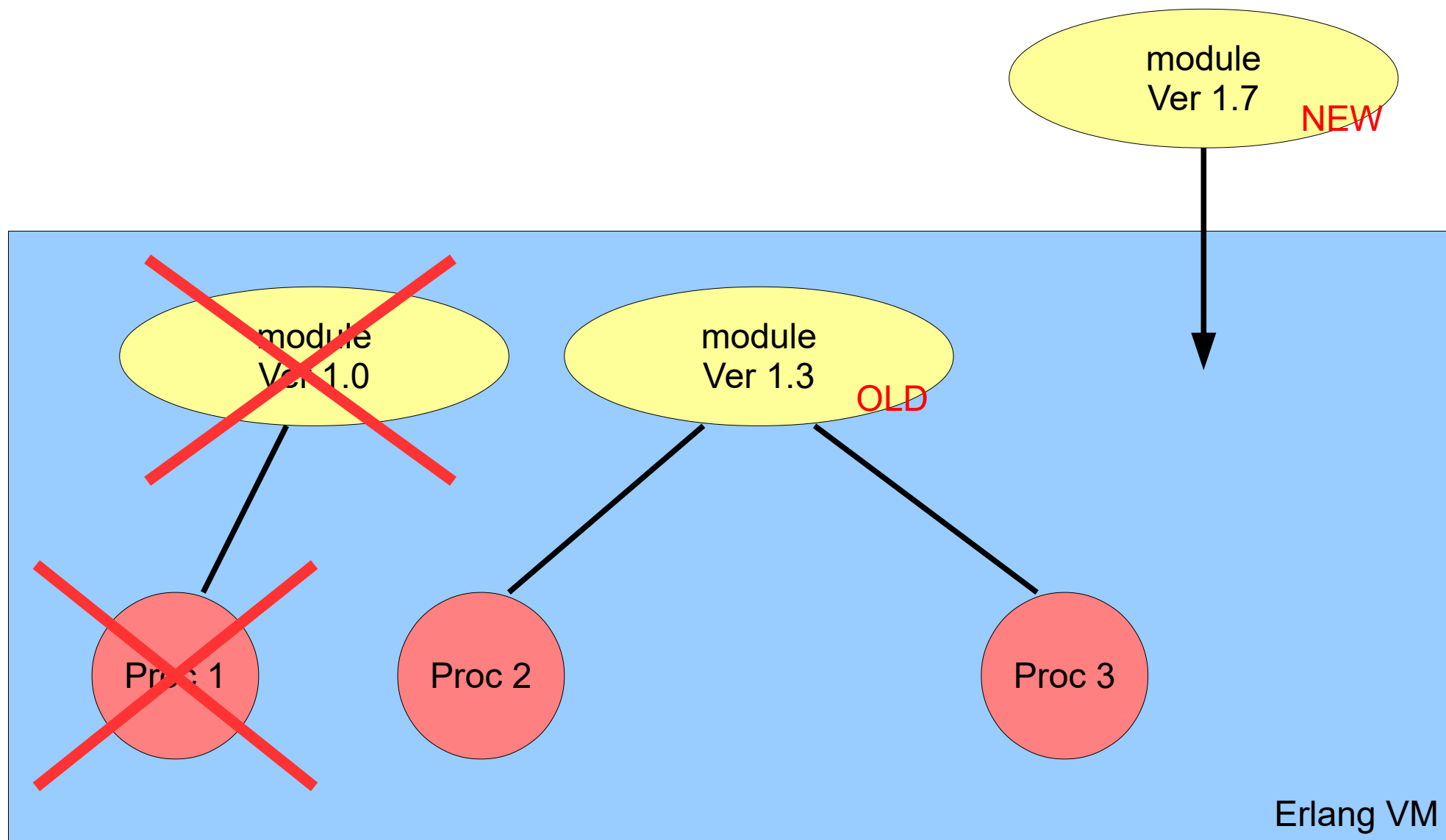
Wersje modułów



Wersje modułów



Wersje modułów



Kontrolowane modyfikowanie modułów

- Code Server pozwala na usunięcie starej wersji modułu w sposób kontrolowany
- *code:purge(moduleName)* - usuwa starą wersję modułu, usuwa wszystkie procesy używające starej wersji modułu
- *code:soft_purge(moduleName)* - usuwa starą wersję modułu jeśli żaden proces z niej nie korzysta; zwraca *true*, jeśli stara wersja została usunięta

- Code Server ładuje moduły, gdy:
 - Wywołana zostanie funkcja z modułu, który nie jest załadowany
 - Moduł zostanie skompilowany: *c(modulename)*
 - Moduł zostanie załadowany w shellu: *l(fileName)*
 - Wywołana zostanie funkcja *code:load_file(fileName)*
- Code Server przeszukuje ustawione ścieżki w poszukiwaniu modułów

```
33> code:get_path().  
[".", "c:/PROGRA~1/ERL57~1.5/lib/kernel-2.13.5/ebin",  
  "c:/PROGRA~1/ERL57~1.5/lib/stdlib-1.16.5/ebin",  
  "c:/PROGRA~1/ERL57~1.5/lib/xmerl-1.2.4/ebin",  
  
34> code:add_path("c:/moduseDir").
```

Programowanie rozproszone

- Wydajność
- Rozszerzalność i skalowalność
- Dostępność
- Udostępnianie zdalnych zasobów
- Wykorzystanie heterogenicznych technologii

- Węzeł - instancja maszyny wirtualnej Erlanga
- By mógł komunikować się z innymi maszynami Erlanga, musi posiadać unikatowy **identyfikator** czyli **nazwę**
- Zwyczajowo identyfikator ma postać *Name@Host*
- Nadanie identyfikatora/nazwy węzła jest możliwe podczas uruchamiania maszyny - parametr:
 - sname *Nazwa* lub
 - name *PelnaNazwa*

- Ciasteczko - identyfikator grupy węzłów
- Tylko węzły z tym samym ciasteczką mogą się komunikować
- Ciasteczko nadaje się przy uruchamianiu węzła opcją *-setcookie Ciasteczko*
- Domyślne ciasteczko dla danego komputera jest losowane i zapisywane w pliku *~/.erlang.cookie*
- Czyli domyślnie wszystkie węzły Erlanga na tej samej maszynie mogą się komunikować

- Węzły łączą się same, przy pierwszej okazji:
 - Uruchomienie zdalnego procesu
 - Jakakolwiek komunikacja z węzłem
- Węzły tworzą siatkę połączeń - każdy z każdym
- ... co można wyłączyć flagą *-connect_all false*
- Testowanie połączenia: *net_adm:ping(Node)*

- Erlang wykorzystuje do nasłuchiwania port 4369
- Protokół nie jest bezpieczny - w sieciach zdalnych trzeba stosować tunelowanie

- BIF:
 - node/0 - nazwa bieżącego węzła
 - node/1 - nazwa węzła, na którym znajduje się proces
 - nodes/0 - lista podłączonych węzłów

- Węzły, które nie łączą się automatycznie z innymi węzłami
- Trzeba łączyć je ręcznie z każdym innym węzłem
- Wykorzystywane do zarządzania, nadzorowania, śledzenia...
- Uruchamianie: flaga *-hidden*

Uruchamianie zdalnych procesów

- `Pid = spawn(NodeName, Module, Function, Arguments)`
- `Pid = spawn_link(NodeName, Mod, Fun, Arguments)`
- Kod musi być dostępny na węźle zdalnym - Erlang sam nie dystrybuuje kodu pomiędzy węzłami

- Funkcja *register/2* działa tylko lokalnie
- Moduł *global*
- *register_name(Name, Pid, Resolve) -> yes | no*
- *registered_names() -> [Name]*
- *unregister_name(Name) -> void()*
- *whereis_name(Name) -> pid() | undefined*
- *send(Name, Msg) -> pid()*

Przykłady

- *Na laboratorium...*

Erlang: The Movie

Erlang The Movie II: The Sequel

