Pracownia z Kursu języka Erlang
Lista 6

W poniższych zadaniach, jeśli to możliwe, należy używać zachowania `gen_server`. Raczej niezbędne jest także użycie modułów `rpc`, `error_logger`, `global`, `net_adm`. Zadania należy oddawać na pracowni w jawny sposób korzystając z wielu komputerów. Oddawanie zadania na laptopie jest niedopuszczalne.

Komputery w pracowni dzielą system plików przechowujący katalog domowy. Dla wygody pracy z modułem `slave` wygeneruj prywatny i publiczny klucz `SSH`. Klucz publiczny zapisz do pliku `.authorized_keys`, a prywatny dodaj do repozytorium aktywnych kluczy przy pomocy polecenia `ssh-add`. Możliwe, że najpierw będziesz musiał wystartować proces `ssh-agent`. Więcej informacji na temat autoryzacji za pomocą kluczy znajdziesz w podręczniku do *OpenSSH*.

1. **(1.5 pkt)** Przygotuj proces, którego zadaniem będzie zarządzanie węzłami. Węzły mają być tworzone i usuwane przy pomocy modułu `slave`. Pierwszym argumentem niżej wymienionych funkcji jest nazwa procesu, która powinna być również użyta do konstrukcji nazwy węzła (`nazwa_procesu@komputer`):

   - `start/1` pobierze listę komputerów z pliku `.hosts.erlang` (patrz moduł `net_adm`); wystartuje nazwany proces, który odpali na wskazanych komputerach węzły;
   - `start/2` j.w. przy czym przyjmuje listę hostów jako drugi parametr;
   - `stop/1` zamknie wszystkie węzły i zakończy proces;
   - `add_host/2` (`cast`) prześle do procesu nową nazwę komputera, na którym ma być odpalony węzeł;
   - `remove_host/2` (`call`) prześle do procesu nazwę komputera, na którym chcemy zakończyć działanie węzła;
   - `choose_node/1` (`call`) zwraca najmniej obciążony w bieżącej chwili węzeł; obciążenie należy wyliczyć korzystając z funkcji `cpu_sup:util/0`; zanim jej użyjesz należy wystartować na zdalnej maszynie aplikację `sasl` i `os_mon`; być może przyda się wywołanie `rpc:parallel_eval`.

     ```
     > application:start(sasl).
     > application:start(os_mon).
     > cpu_sup:util().
     5.7578129583130035
     ```

   Należy monitorować zdalne maszyny. Jeśli, któraś z nich padnie, należy spróbować ją podnieść. Jeśli się to nie uda należy zalogować, że operacja się nie powiodła i porzucić jej monitorowanie.

   Przygotuj prezentację zadania. Pokaż, co się stanie jeśli na jednej z maszyn zabijesz proces `erl` poleceniem `kill`.

2. **(2 pkt)** Należy zaimplementować rozproszone sortowanie przez scalanie i mierzyć wydajność jego działania. Przygotuj zestaw danych np. plik z listą kilkunastu milionów liczb (`file:consult/1` może się przydać przy wczytywaniu). Moduł powinien implementować jedną funkcję, która rozproszy obliczenia na listę węzłów.

   ```
   parallel_merge_sort(Nodes,ProcPerNode,ListToSort) -> {SortedList,TimeInMiliSecs}
   ```

   Należy sprawiedliwie rozdzielić pracę, równo między wszystkie węzły w klastrze. Każdy węzeł startuje określoną ilość procesów, a następnie między nie dzieli pracę. Po wykonaniu pracy przez procesy dokonujemy łączenia wyników, najpierw lokalnie dla węzła, a potem globalnie.

   Przygotuj prezentację – zmierz i zaloguj czas działania poszczególnych faz algorytmu (dzielenie i scalanie dla poszczególnych węzłów oraz procesów). Po skończeniu działania wyświetl ile czasu zajęło działanie sortowania. Jak dobrze skaluje się sortowanie przez scalanie – t.j. jaki wpływ na szybkość obliczeń ma ilość węzłów?

3. **(3 pkt)** Należy zaimplementować proces implementujący interfejs słownika `dict`. Proces musi posiadać globalną nazwę i na żądanie będzie migrował na wskazany węzeł. Migracja musi być przezroczysta – nie dopuszczamy utraty jakiegokolwiek komunikatu. Rozwiązanie może wyglądać następująco:

   Po otrzymaniu komunikatu `{migrate,Node}` proces `Pid1` wystartuje na zdalnej maszynie drugą instancję samego siebie `Pid2` i przekaże jej własny słownik oraz identyfikator. `Pid2` rejestruje się w globalnym systemie nazw, a następnie czeka na wyjście procesu `Pid1` (sygnał `{'EXIT',Pid1,normal}`). `Pid1` dostaje sygnał o konflikcie od globalnego systemu nazw. Przekazuje wszystkie nieobsłużone komunikaty ze swojej skrzynki pocztowej do `Pid2`, a następnie kończy swoje działanie. `Pid2` reaguje na komunikat o wyjściu starego procesu i wznawia obsługę żądań.

   Przygotuj prezentację pokazującą, że Twoje rozwiązanie działa pod obciążeniem – np. migruj proces w trakcie dodawaj do słownika elementów z kluczem o rosnących liczbach naturalnych.

   Podpowiedź: oprócz logiki przekazywania sterowania resztę należy zrobić możliwie jak najprościej. Wystarczy proste proxy, które jako żądanie synchroniczne będzie brało nazwę funkcji z modułu `dict` *oraz listę jej argumentów.*

```
1> Module = dict.
2> SomeFunc1 = new.
3> D0 = Module:SomeFunc1().
4> SomeFunc2 = append.
5> D1 = Module:SomeFunc2(a,10,D0).
6> dict:fetch_keys(D1).
[a]
```

4. **(0.5 pkt)** Jeśli robimy zdalne synchroniczne wywołanie z ograniczeniem czasowym, to możemy natrafić na subtelny problem. Załóżmy, że pierwsze żądanie wysłane do serwera się przedawniło. Klauzula `after` wygeneruje krotkę `{error,timeout}`. Następnie, być może z innymi danymi, ponawiamy żądanie. Serwer zaczyna odpowiadać i przysyła na odpowiedź na poprzednie i bieżące zapytanie. Jak sobie poradzić z tym problemem? Zaimplementuj odpowiedni moduł (nie wolno używać biblioteki `rpc`) z funkcjami:

   - `send(Name,Request,Timeout) -> Response | {error,Reason}`
   - `send(Node,Name,Request,Timeout) -> Response | {error,Reason}`

   Przygotuj prosty proces do prezentacji zadania, który na pierwsze żądanie będzie odpowiadał z dużym opóźnieniem.

   Podpowiedź: wspominałem o tym krótko na wykładzie – trzeba skonstruować prosty proces pośredniczący, który zginie jeśli żądanie się przedawni. Serwer przedawnioną odpowiedź prześle do nieistniejącego procesu czyli do `/dev/null`*. Można dodawać też referencje do wywołań, ale nie jest to już przezroczyste.*

Lista i materiały znajdują się pod adresem
http://cahirwpz.cs.uni.wroc.pl/main-pl/erlang-language-summer-2010/ *Krystian Bacławski*