Paweł Kamiński
Jan Muczyński
Jakub Guzek
Tomasz Bocheński

Dokumentacja końcowa UXP1A
Potoki nienazwane

1. Treść zadania

Napisać wieloprocesowy program realizujący komunikację w języku komunikacyjnym Linda. W uproszczeniu Linda realizuje 3 operacje:

output(krotka)
input(wzorzec_krotki, timeout)
read(wzorzec_krotki, timeout)

Komunikacja międzyprocesowa w Lindzie realizowana jest poprzez wspólną dla wszystkich procesów przestrzeń krotek. Krotki są arbitralnymi tablicami dowolnej długości składającymi się z elementów 3 typów podstawowych: **string, integer, float**. Przykłady krotek: (1, “abc”, 3.1415, “d”), (10, “abc”, 3.1415) lub (2,3,1, “Ala ma kota”). Funkcja **output** umieszcza krotkę w przestrzeni. Funkcja **input** pobiera i atomowo usuwa krotkę z przestrzeni, przy czym wybór krotki następuje poprzez dopasowanie wzorca-krotki. Wzorzec jest krotką, w której dowolne składniki mogą być niewyspecyfikowane: “ * ” (podany jest tylko typ) lub zadane warunkiem logicznym. Przyjąć warunki: ==, <, <=, >, >=. Przykład: **input (integer:1, string:*, float:*, string:”d”)** - pobierze pierwszą krotkę z przykładu wyżej zaś: **input(integer:>0, string:”abc”, float:*, string:*)** drugą. Operacja **read** działa tak samo jak **input**, lecz nie usuwa krotki z przestrzeni. Operacje **read** i **input** zawsze zwracają jedną krotkę. W przypadku gdy wyspecyfikowana krotka nie istnieje operacja **read** i **input** zawieszają się do czasu pojawienia się oczekiwanej danej.

2. Interpretacja treści zadania

Zadanie polega na stworzeniu komunikacji międzyprocesowej korzystając z potoków nienazwanych. Aby skorzystać z takiego rodzaju komunikacji, procesy muszą być spokrewnione, dlatego w tym celu procesy będziemy tworzyć z wykorzystaniem funkcji fork(). Procesy utworzone dzięki tej funkcji, będą dziedziczyć deskryptory, a co za tym idzie będą miały dostęp do potoków nienazwanych, służących do ich komunikacji z procesem macierzystym. Program będzie umożliwiał operowanie na wspólnej przestrzeni krotek wszystkim procesom potomnym(‘klientom’), które będą wysyłały polecenia zapisane w języku komunikacyjnym Linda, do procesu

macierzystego, pełniącego rolę ‘serwera’. Polecenia będą przekazywane przez potoki nienazwane. Każdy proces potomny będzie dysponował pojedynczym potokiem nienazwanym, przypisanym jemu i tylko jemu, poprzez który będzie otrzymywał odpowiedzi od procesu macierzystego, wykonującego polecenia procesów potomnych. Polecenia te, procesy potomne, będą zapisywać do wspólnego potoku nienazwanego, gdyż założeniem naszego projektu jest zasada, iż procesy potomne, pełniące rolę ‘klientów’, będą przesyłały polecenia w języku Linda, z wykorzystaniem tylko jednego potoku. Dzięki temu, proces macierzysty będzie mógł czytać polecenia tylko z jednego potoku zaś wyniki będzie zapisywał do odpowiedniego potoku, dedykowanego dla określonego ‘klienta’. Dzięki takiemu schematowi komunikacji, nie będziemy musieli dbać o to, aby ‘serwer’ sprawiedliwie obsługiwał ‘klientów’, gdyż będzie po prostu realizował te polecenia, które znajdą się w potoku.

3. Szczegółowy opis funkcjonalny - proponowane API

Funkcjonalność naszego programu będzie się opierała na trzech głównych poleceniach z języka komunikacyjnego Linda, operujących na należących do tej samej przestrzeni krotkach:

- input(const TuplePattern& pattern, unsigned timeout, Tuple& returnTuple)
- output(const Tuple& tuple)
- read(const TuplePattern& pattern, unsigned timeout, Tuple& returnTuple)

TuplePatern - klasa reprezentująca wzorzec krotki, który ma spełniać krotka zwracana w odpowiedzi na polecenia input oraz read.
Tuple - klasa agregująca. Przechowuje kolejne elementy 3 podstawowych typów.

bool input(**const** TuplePattern& pattern, **unsigned** timeout, Tuple& returnTuple);
Funkcja, która na podstawie podanych argumentów generuje wiadomość, która jest reprezentowana przez obiekt klasy Message. Wiadomość jest przed wysłaniem serializowana. Następnie, w takiej formie, wiadomość jest przekazywana z wykorzystaniem potoku nienazwanego.

bool output(**const** Tuple& tuple);
Funkcja, przy pomocy której, można dodać podaną w argumencie krotkę, do wspólnej przestrzeni krotek. Podobnie jak w funkcji input, tworzona jest wiadomość, która później podlega serializacji i następnie zostaje przekazana za pomocą potoku.

bool read(**const** TuplePattern& pattern, **unsigned** timeout, Tuple& returnTuple);
Funkcja, która umożliwia odczytanie krotki spełniające warunki opisane we wzorcu przekazanym w argumencie wywołania. Kolejno, jak w przypadku poprzednich

funkcji, zostaje utworzona wiadomość, następnie jest serializowana i wysyłana przy pomocy odpowiedniego potoku.

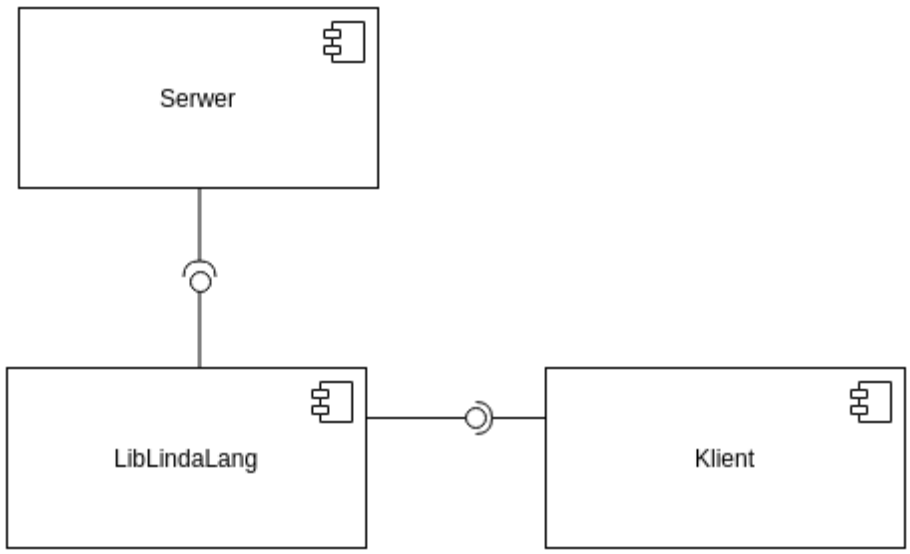
Wszystkie funkcje wyżej opisane, są zgrupowane w obrębie jednej klasy Linda. Klasa ta wykorzystuje do komunikacji dwa potoki. Jeden, służący do wysyłania poleceń oraz drugi, służący do przekazywania odpowiedzi i tak funkcje input oraz read korzystają z jednego potoku zaś funkcja output korzysta z drugiego potoku. Potoki potrzebne do realizacji funkcji wyżej opisanych są przekazywane w argumentach konstruktora obiektu klasy Linda. Klient musi sam stworzyć instancje klasy Pipe, które są w module Linda.

Linda(const Pipe& pipeResponse, const Pipe& pipeRequest)
Konstruktor klasy Linda, którego argumentami są dwa obiekty klasy Pipe, które reprezentują potok jak i również udostępniają funkcje read i write (oraz kilka innych niezbędnych do poprawnego korzystania z potoku, jak np. checkReadingAvailibility umożliwiająca sprawdzenie, czy w potoku znajdują się dane do odczytu czy nie) dla potoku.

4. Podział na moduły/procesy i struktura komunikacji między nimi, realizacja współbieżności

Moduły:

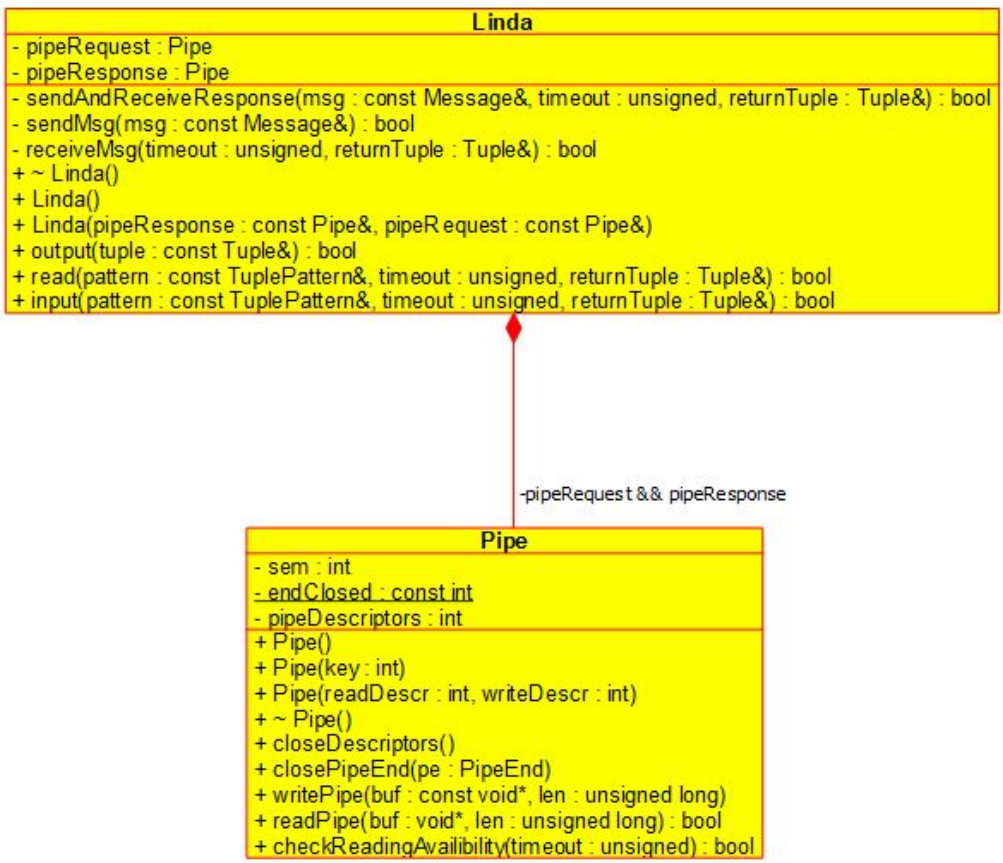
1) LindaLang - biblioteka implementująca operacje w języku Linda. Dostarcza implementację API realizującego główną funkcjonalność.
2) Serwer - proces stanowiący repozytorium krotek. Jego zadaniem jest:
- przyjmowanie krotek wysyłanych przez klientów;
- wysyłanie krotek spełniających odpowiednie wymaganie klientom.
3) Klient - proces którego zadaniem jest:
- tworzenie i wysyłanie krotek do serwera;
- pobieranie lub odczytywanie krotek z serwera.

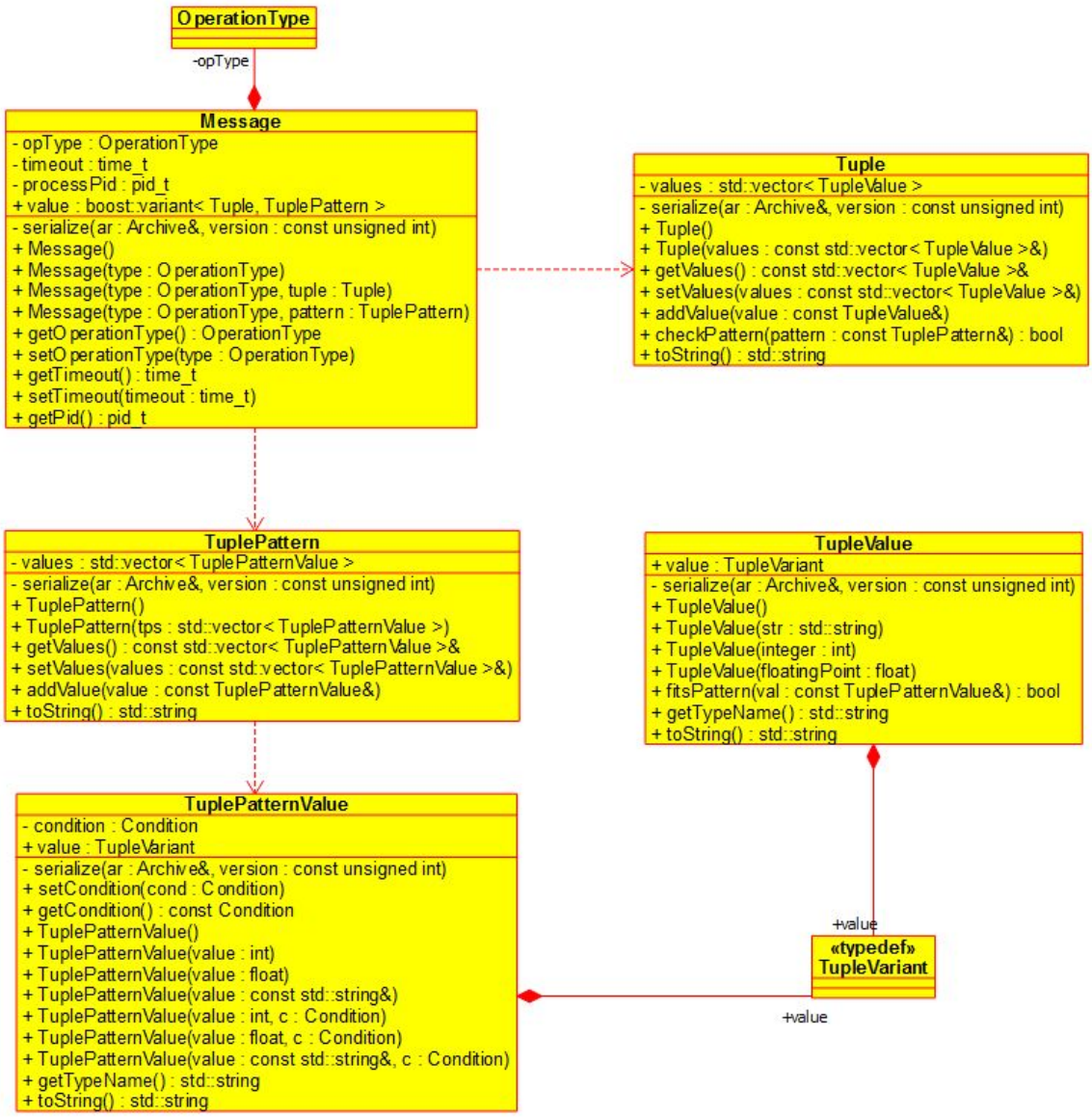


Klasy składające się na poszczególne moduły:

1. LibLindaLang:
    a. ConditionVis - klasa wizytatora służąca do porównywania krotek
    b. ConditionTraits -
    c. Linda - klasa służąca do komunikacji klienta z serwerem przez potoki nienazwane. Umożliwia wysyłanie krotek oraz ich pobieranie z globalnej przestrzeni.
    d. Message - klasa reprezentująca wiadomość przesyłaną potokiem nienazwanym przez klienta na serwer.
    e. LindaLogger - klasa służąca do zapisywania do plików logów. Są cztery poziomy logów: ERROR, WARNING, INFO oraz DEBUG.
    f. Pipe - klasa umożliwiająca komunikację przez potoki nienazwane.
    g. Tuple - klasa reprezentująca pojedynczą krotkę. Posiada kontener obiektów TupleValue oraz metodę do sprawdzenia dopasowania krotki z podanym wzorcem TuplePattern.
    h. TupleGenerator - klasa (singleton) udostępniająca metody generujące wzorce krotek (TuplePattern) oraz krotki (Tuple)
    i. TuplePattern - klasa reprezentująca wzorzec krotki, posiada kontener obiektów TuplePatternValue.
    j. TuplePatternValue - klasa reprezentująca pojedynczą wartość we wzorcu krotki, składa się z operacji na niej wykonywanej wykonywanej oraz wartości np integer: >12
    k. GetTypeNameVis - klasa implementująca wzorzec wizytatora, wykorzystywana do implementacji metody identyfikującej typ elementu krotki
    l. ToStringVis - klasa implementująca wzorzec wizytatora, wykorzystywany do implementacji metody toString dla elementów krotek
    m. TupleValue - klasa reprezentująca pojedynczą wartość w krotce

2. Serwer:
    a. Server - klasa używana przez proces macierzysty, serwer czyta wiadomości z potoku zapytań i następnie je przetwarza i w razie potrzeby wysyła odpowiedzi do klientów.
    b. TupleSpace - globalna przestrzeń krotek, posiada metody do dodawania krotek do przestrzeni oraz ich usuwania i wyszukiwania po wzorcu.
3. Klient:
    a. Client - klasa reprezentująca klienta, korzysta ze wspólnego dla wszystkich potoku zapytań i posiada swój własny potok odpowiedzi z serwera. Klasa jest inicjalizowana poprzez podanie do jej konstruktora obu wspomnianych powyżej potoków i posiada metodę, która w pętli nieskończonej generuje i wysyła zapytania do serwera oraz odbiera odpowiedzi.





5. Opis najważniejszych rozwiązań funkcjonalnych wraz z uzasadnieniem (opis protokołów, kluczowych funkcji itp.)

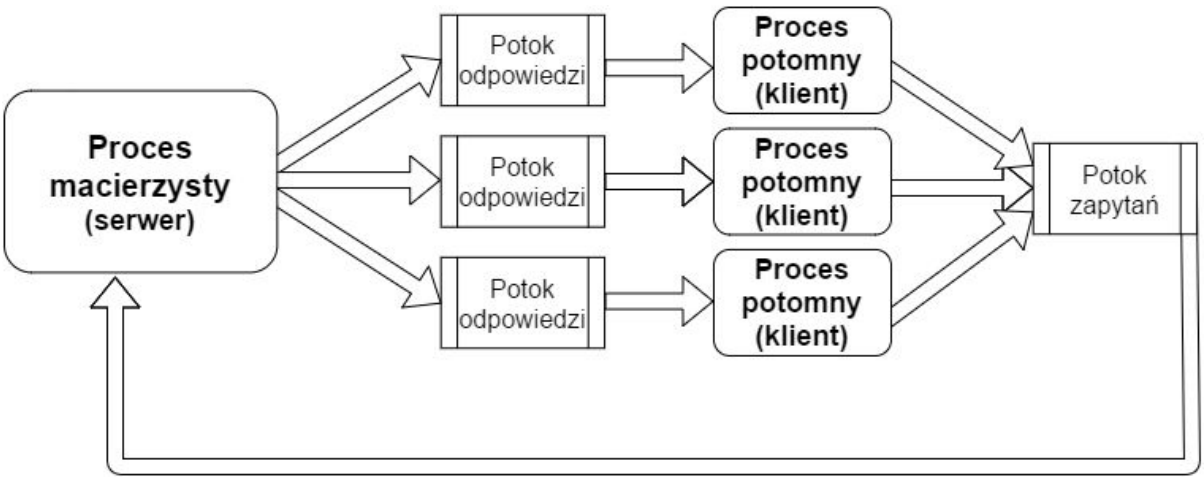
- **Krotka**
  Podstawową strukturę projektu, jaką jest pojedynczy element krotki, zaimplementowaliśmy z wykorzystaniem variant'u z biblioteki boost. Uznaliśmy, że jest to najlepszy i najwygodniejszy sposób implementacji elementu krotki, ponieważ dzięki takiemu rozwiązaniu uniknęliśmy wielokrotnego implementowania różnych szablonów. Wszelkie metody dla pojedynczego elementu krotki zostały zaimplementowane z wykorzystaniem wzorca projektowego wizytatora. Krotka zatem jest reprezentowana w postaci

wektora variant'ów.

- **Potok**
  Potoki wraz z wszelkimi dostępnymi na nich funkcjami, są reprezentowane w postaci klasy. Klasa korzysta z systemowych potoków nienazwanych. W trosce o dobrą synchronizację dostępu do potoku tj. aby mieć pewność, że w potoku znajduje się tylko jedna wiadomość, klasa Pipe korzysta również z systemowych semaforów nazwanych (powyższe ograniczenie jest podyktowane faktem, iż przy obecnym sposobie konstruowania wiadomości nie istnieje możliwość identyfikacji wiadomości oraz jej długości).

- **Message**
  Wiadomości, które są przekazywane za pomocą potoków nienazwanych, są reprezentowane za pomocą klasy Message. Klasa ta agreguje krotkę bądź jej wzorzec, rodzaj operacji do wykonania, timeout oraz pid. Takie wiadomości są serializowane dzięki wykorzystaniu boost::serialization.

6. Architektura i komunikacja



Serwer zajmuje się magazynowaniem krotek. Ponieważ wszyscy klienci muszą mieć możliwość komunikowania się z serwerem, jest on rodzicem wszystkich klientów. Dla każdego nowego procesu(klienta) tworzonego przez serwer otwierany jest jeden potok do odczytywania odpowiedzi z serwera. Klient ma również dostęp do potoku zapytań, w którym umieszcza zapytania do procesu macierzystego.

Komunikaty wysyłane do serwera są zserializowanymi obiektami klasy Message, która posiada takie pola jak:

- OperationType opType - enum przybierający jedną z możliwości OUTPUT, INPUT lub READ, które mówią jaka akcja ma zostać wykonana na przestrzeni krotek.
- pid_t processPid - pid procesu wysyłającego zgłoszenie, podawany w celu poinformowania serwera do kogo będzie miał skierować odpowiedź.
- boost::variant<Tuple, TuplePattern> value - przyjmuje jedną z wartości typu Tuple lub TuplePattern, które reprezentują odpowiednio krotkę lub jej wzorzec, który zostaje wysłany na serwer
- time_t timeout - czas przedawnienia się wiadomości serwera. Po jego upływie serwer nie odsyła odpowiedzi do klienta.

Zapis i odczyt do potoków jest synchronizowany poprzez semafory nazwane, które uniemożliwiają zapis innym klientom, dopóki wiadomość nie zostanie odebrana przez serwer. Służy to zapobiegnięciu sytuacji podczas której serwer przypadkowo odebrałby dwie wiadomości i przeczytał je jako jedną, co spowodowałoby niepoprawną deserializację.

7. Postać logów

Na logi w projekcie składają się:
1. Logi Klientów
2. Logi Serwera
3. Logi Modułu Linda

Ad 1. Logi Klientów
Każdy klient ma swój plik z logami, który jest nazwany “client” + pid klienta + “.log”, na przykład “client3055.log”. Mają one postać ‘***timestamp’*: ‘wiadomość’**.
Timestamp to sformatowany znacznik czasu, w którym wystąpiło zdarzenie.
Zdarzenie jest opisane w wiadomości. Wiadomości są typu:
- sent tuple
- sent TuplePattern to read
- received Tuple
- sent TuplePattern to input
Przykładowe logi:
2016-06-01 23:28:40: connected
2016-06-01 23:28:40: sent tuple (integer:*-674983652*,integer:*-1374670078*,integer:*-1484008502*,float:*-32.296936*)
2016-06-01 23:28:42: sent TuplePattern to read (integer:>*-2039253235*,string:"C3EJK41JH9C8JAC3",integer:<=*-947219146*)
2016-06-01 23:28:44: sent TuplePattern to input(float:*,integer:*1051367524*)
2016-06-01 23:28:54: received Tuple (float:*37.744583*,float:*88.021439*)

Ad 2. Logi serwera
Jest jeden serwer i tworzy on logi w pliku “server” + pid serwera + “.log”, na przykład “server3054.log”. Mają postać taką, jak klient, czyli ‘***timestamp’*: ‘wiadomość’**.
Wiadomości są typu:
- received Tuple
- received TuplePattern
- found tuple
- sent Tuple

Przykładowe logi:
2016-06-01 23:*28*:*40*: received Tuple (integer:*734920499*,integer:*254964351*,integer:*179184265*)
2016-06-01 23:*28*:*40*: received TuplePattern (integer:>=*573294037*,integer:*,integer:>=*832770510*,integer:*)
2016-06-01 23:*28*:*40*: found tuple
(integer:*2136215699*,integer:*1079785500*,integer:*1007516198*,integer:*2056156631*)
2016-06-01 23:*28*:*40*: sent Tuple
(integer:*2136215699*,integer:*1079785500*,integer:*1007516198*,integer:*2056156631*)

Ad 3. Logi Modułu Linda
Logi modułu Linda są zapisywane zapisane w pliku linda.log. Mają postać taką, jak klient, czyli ‘***timestamp’* ‘rodzaj wiadomości’: pid ‘pid procesu, który wywołał zdarzenie’ ‘opis zdarzenia’**. Timestamp to sformatowany znacznik czasu. Rodzaj wiadomości to jeden z **DEBUG**, **INFO**, **ERROR**.

Przykładowe logi:
2016-06-01 23:*28*:*40* DEBUG: pid *3054* Pipe() key = *1*
2016-06-01 23:*28*:*40* INFO: pid *3054* closePipeEnd() ReadEnd
2016-06-01 23:*28*:*40* DEBUG: pid *3058* Linda()
2016-06-01 23:*28*:*40* DEBUG: pid *3058* output() entry:
(integer:*-444516812*,float:*24.794655*,float:*-34.587067*,string:"E4K7F9")
2016-06-01 23:*32*:*47* INFO: pid *3054* checkPattern() Tuple
(string:"1DF4C5IFF9CI1D",integer:*-896191492*,float:*-56.175076*) matches pattern
(string:<"1E808AFHK1757AAA",integer:*,float:<*27.269653*)
2016-06-01 23:*32*:*47* DEBUG: pid *3054* writePipe()
2016-06-01 23:*32*:*47* INFO: pid *3054* checkPattern() Tuple
(string:"1DF4C5IFF9CI1D",integer:*-896191492*,float:*-56.175076*) matches pattern
(string:<"1E808AFHK1757AAA",integer:*,float:<*27.269653*)

8. Opis wykorzystywanych narzędzi

Język implementacji - C++11
Używane biblioteki: STL oraz boost
Narzędzia wspomagające budowanie: CMake

9. Opis metodyki testów i wyników testowania

W projekcie zostały użyte dwie metody testowania.

1) Pierwsza z metod to testy jednostkowe. Jest ich około dwudziestu. Pokrywają one najważniejsze funkcjonalności biblioteki.
2) Druga z metod, których użyliśmy do testowania to sprawdzenie biblioteki w praktyce. Zostały napisane dwa moduły: Client oraz Server. Działa to w następujący sposób: została stworzona instancja klasy Pipe, jako pipeRequest. Następnie tworzymy n (w teście n=5) procesów potomnych. Dla każdego procesu tworzymy instancję klasy Pipe - pipeResponse (odpowiedź) i tworzymy instancje klienta, korzystając z jego konstruktora - *Client(**const Pipe&** pRequest, **const Pipe&** pResponse)* podając pierwszy argument - pipeRequest (taki sam dla każdego klienta), a drugi pipeResponse (inny dla każdego klienta). Następnie dla każdego klienta wywołujemy metodę run(), która w nieskończonej pętli generuje nowe zapytania do serwera. Następnie zamykamy niepotrzebne deskryptory. Proces macierzysty tworzy instancje klasy Server, używając kontruktora *Server(**const Pipe&** p);* podając jako argument pipeRequest. Powinniśmy przekazać serwerowi mapę (Klucz - pid procesu; wartość - deskryptor Pipe) utworzonych Pipe'ów. Kolejno dodajemy m (w teście m=1000) tupli do servera, aby mieć jakąś bazę. Na sam koniec wywołujemy metodę serwera processRequests(), która w nieskończonej pętli odczytuje wiadomości z potoku nienazwanego i następnie je przetwarza - np. umieszczenie krotki w przestrzeni, znalezienie krotki pasującej do wzorca i

wysłanie jej do klienta.