Computational Cluster

Katarzyna Węgiełek
Paweł Własiuk
Kamil Sienkiewicz
Marcin Wardziński

30 stycznia 2015

Spis treści

Rozdział 1

Wstęp

Tematem projektu jest stworzenie dokumentacji dla klastra obliczeniowego. Zadaniem projektowanego przez nas systemu będzie prowadzenie obliczeń rozproszonych. Klaster będzie umożliwiał rozwiązywanie skomplikowanych problemów, wykorzystujących algorytmy o dużej złożoności czasowej, w szczególności rzędu $o(2^n)$. Jego najważniejszą rolą będzie odpowiednie rozdzielenie zadań pomiędzy różne elementy systemu tak, aby jak najbardziej optymalnie wykorzystywać moc obliczeniową komputerów, z których się składa. Istotne jest również zminimalizowanie ryzyka utraty danych w przypadku awarii któregoś z komponentów.

1.1 Elementy klastra obliczeniowego

Task manager - dzieli zadanie na podproblemy i oblicza rozwiązanie końcowe na podstawie rozwiązań częściowych

Computational node - rozwiązuje pojedynczy podproblem i odsyła do jego rozwiązanie częściowe do communications server'a

Communications server - przesyła podproblemy i rozwiązania częściowe między task manager'ami a computational node'ami oraz wysyła rozwiązanie zadania do computational client'a

Computational client - wysyła problem do communications server'a i oczekuje na otrzymanie rozwiązania

Task solver - moduł używany przez computational node do rozwiązania podproblemu oraz przez task manager do podzielenia zadania i znalezienia ostatecznego rozwiązania na podstawie wyników częściowych

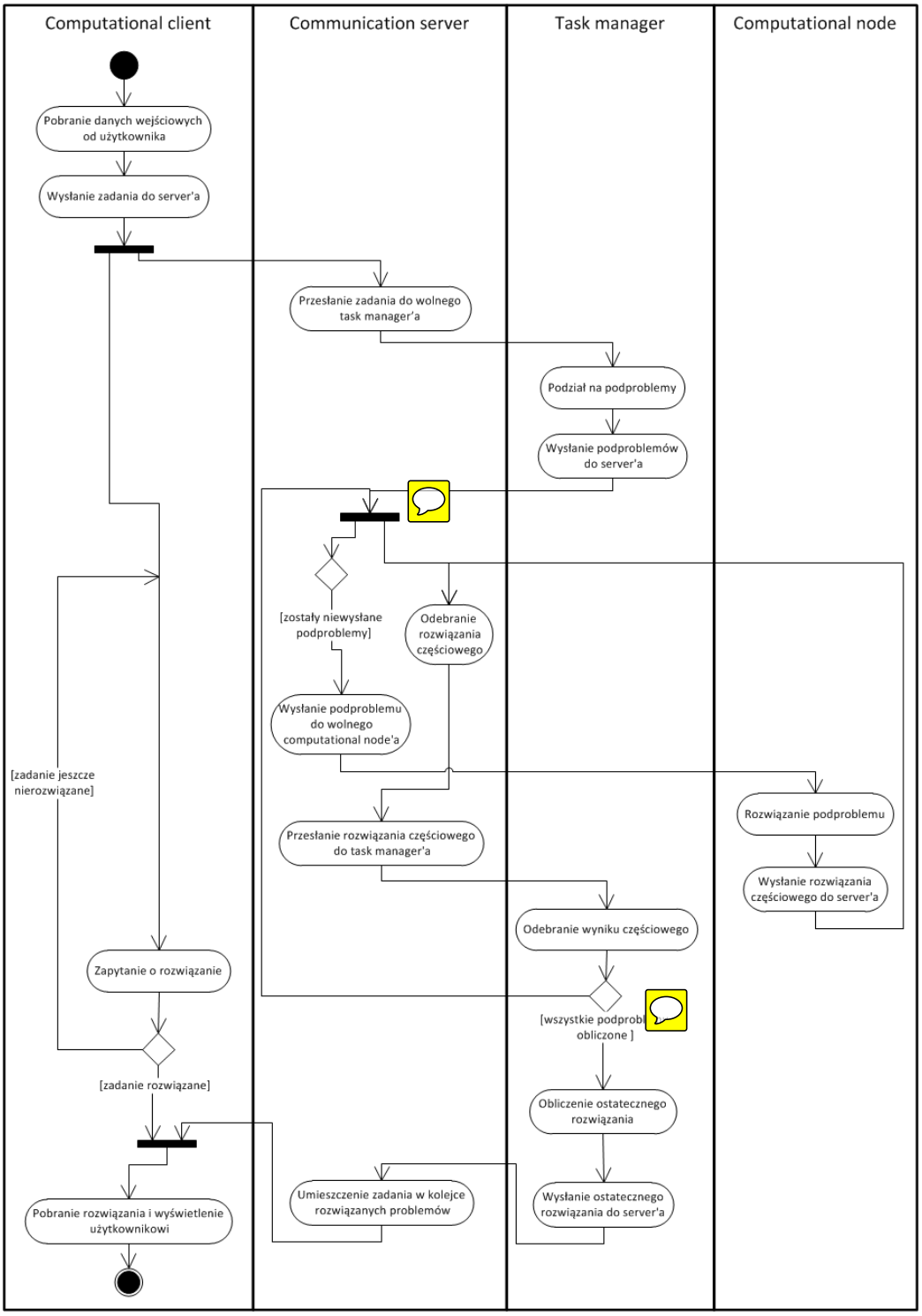
Rozdział 2

Opis działania systemu

Proces rozwiązywania zadania rozpoczyna się od wprowadzenia przez użytkownika danych wejściowych dla problemu. Robi to za pośrednictwem *computational client'a*. Jest to jedyny element klastra, z którym użytkownik ma bezpośredni kontakt. *Computational client* przetwarza wprowadzone informacje na komunikat i wysyła do *communications server'a* w formie pliku XML.
Następnie zadanie trafia do kolejki zgłoszonych problemów. Jeśli istnieje w danej chwili w klastrze wolny *task manager*, który potrafi obsłużyć zadanie tego typu, to *communications server* wysyła do niego otrzymane dane wejściowe. *Task manager* dzieli problem na mniejsze części - podproblemy, przeznaczone do rozwiązania przez pojedyncze *computational node'y* i odsyła je do *server'a*. To czy *task manager* może podzielić dany problem zależy od tego, czy posiada *task solver*, zajmujący się poszukiwaną klasą problemów.
Communications server wysyła odebrane podproblemy kolejno do wolnych *computational node'ów*, które są w stanie je rozwiązać (zawierają odpowiedni *task solver*). *Node'y* prowadzą obliczenia równolegle, a następnie wysyłają uzyskane wyniki do *server'a*. *Communications server* przesyła rozwiązania częściowe do *task managera*, który oblicza na ich podstawie ostateczny wynik. *Task manager* zaczyna scalanie dopiero po otrzymaniu wszystkich rozwiązań częściowych. Gdy uzyska końcowe rozwiązanie, wysyła je do *server'a*. Trafia ono do kolejki zadań oczekujących na odebranie wyników. Kiedy *computational client* wyśle do *server'a* żądanie pobrania rozwiązania, *server* odsyła mu odpowiednie dane.
Podczas całego procesu główny *communications server* synchronizuje trzy-

mane przez siebie dane z *server'em backup'owym*, aby w razie awarii nie stracić części rozwiązań i zgłoszonych zadań. Jeśli *server* główny ulegnie uszkodzeniu i komunikacja z nim będzie niemożliwa, pozostałe komponenty nie przerwą swojej normalnej pracy i zaczną wymieniać komunikaty z *server'em backup'owym*.

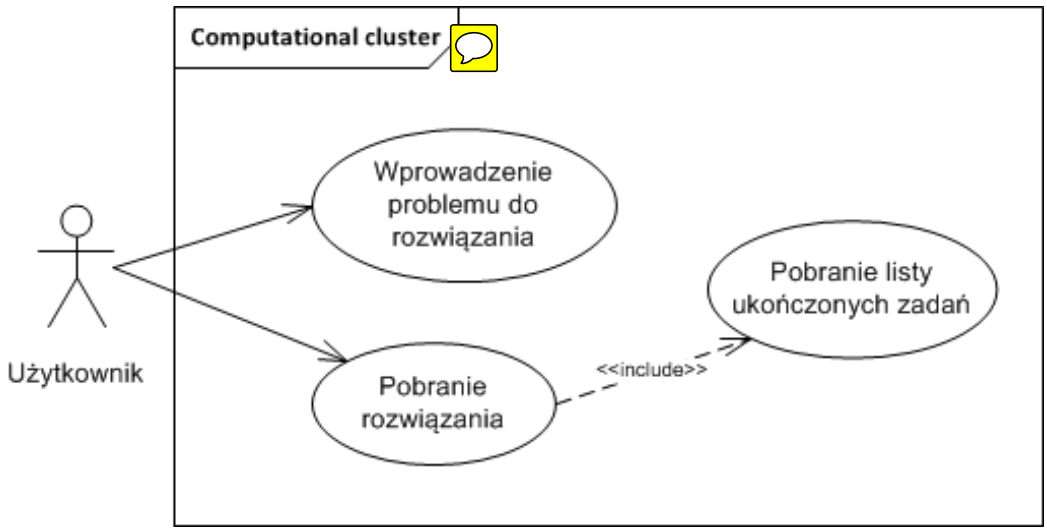
2.1 Diagram aktywności dla rozwiązania pojedynczego problemu



2.2 Diagramy przypadków użycia

2.2.1 Diagram przypadków użycia dla użytkownika

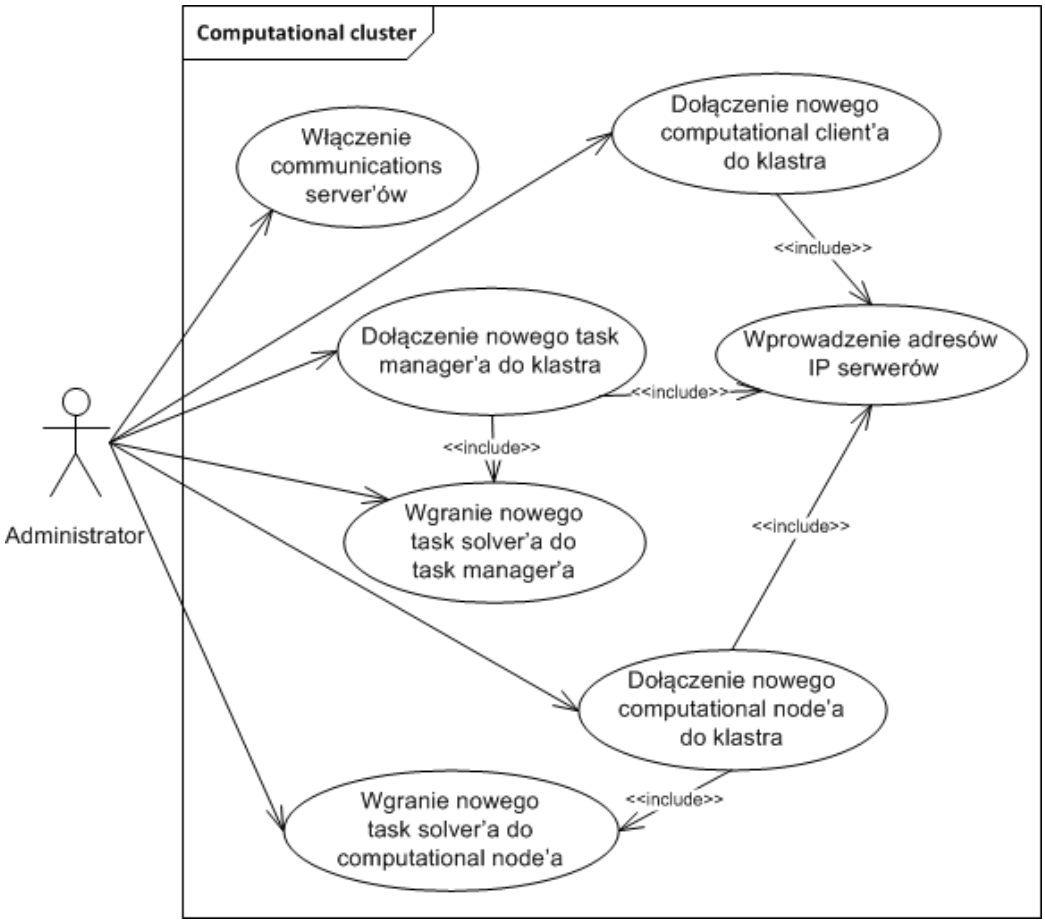
Główną rolą użytkownika jest zdefiniowanie problemu do rozwiązania przez klaster obliczeniowy. Wprowadza on dane wejściowe dla zadania za pośrednictwem *computational client'a*. Gdy obliczenia zostają zakończone, użytkownik pobiera z *communications server'a* rozwiązanie (również za pośrednictwem aplikacji klienckiej). Użytkownik w dowolnym momencie może sprawdzić, czy prace nad jego problemem zostały już zakończone. Na jego żądanie *communications server* wysyła do *computational client'a* listę wszystkich zadań, które zostały już rozwiązane. Jeśli interesujący go problem znajduje się na liście, użytkownik wskazuje identyfikator zadania i pobiera rozwiązanie.



2.2.2 Diagram przypadków użycia dla administratora

Administrator ma za zadanie skonfigurować wszystkie elementy klastra. Najpierw uruchamia *primery communications server* oraz *backup server*. Następnie dołącza do sieci pozostałe komponenty - *task manager'y*, *computational node'y* i *computational client'ów*. Aby umożliwić komunikację pomiędzy elementami systemu, administrator musi zapisać w pliku konfiguracyjnym przyłączanego elementu adresy IP obu serwerów. Żeby *task manager* i *computational node* mogły brać udział w rozwiązywaniu problemu, muszą mieć zainstalowany przynajmniej jeden *task solver*. Zatem przy dołączaniu ich do

klastra administrator powinien zainstalować odpowiednią wtyczkę.
Nowe komponenty mogą być dodawane zarówno przy tworzeniu klastra, jak i już podczas jego działania. Administrator może również w każdym momencie wgrać do *task manager'a* lub *computational node'a* kolejne *task solver'y*, odpowiadające nowym klasom problemów. Zwiększy się w ten sposób ilość typów zadań, jakie klaster jest w stanie rozwiązać.

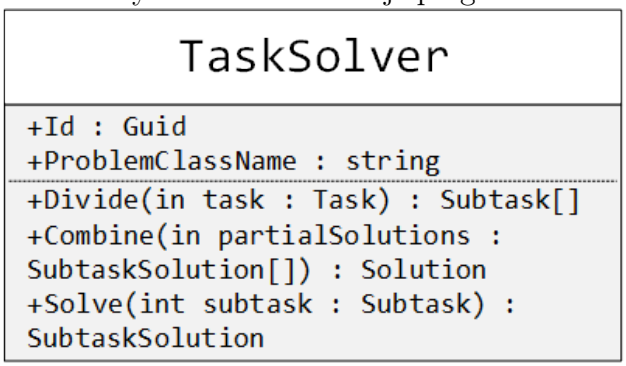


Rozdział 3

TaskSolver

3.1 Interfejs

Rysunek 3.1: Interfejs pluginu



Komponenty systemu odpowiedzialne za wykonywanie obliczeń będą korzystały z interfejsu zaprezentowanego powyżej. Klasa obsługiwana przez system musi implementować 3 metody konieczne do rozwiązania zadania:

- `Divide()` - metoda, która jako parametr przyjmuje *zadanie*. Jej głównym zadaniem jest podzielenie problemu na mniejsze łatwiejsze do rozwiązania podproblemy.
- `Combine()` - metoda, która w parametrze przyjmuje kolekcję częściowych rozwiązań. Jej zadaniem jest scalenie rozwiązań częściowych w jedno ostateczne rozwiązanie.

- `Solve()` - metoda jako parametr przyjmuje podzadanie, które powstało w wyniku działania metody `Divide`. Efektem działania tej metody jest rozwiązanie problemu.

Plugin powinien również udostępniać dwa pola umożliwiające zidentyfikowanie obiektów tej samej klasy w różnych częściach systemu:

- `Id` - unikalne pole typu `Guid`. System nie przewiduje istnienia dwóch różnych klas, rozwiązujących różne zadania identyfikujących się identycznym id. Taka sytuacja mogłaby doprowadzić system do błędu.
- `ProblemClassName` - pole typu `string` zawierające informację zrozumiałą dla użytkownika o typie problemu rozwiązywalnego przez daną klasę.

Rozwiązanie problemu powinno przebiegać w poniżej podany sposób:

(1) zadanie zostaje podzielone na podzadania,

(2) każde z podzadań zostaje rozwiązane,

(3) wszystkie rozwiązania częściowe zostają przekazane jako parametr metody scalającej rozwiązania w wyniku czego otrzymujemy końcowe rozwiązanie naszego problemu.

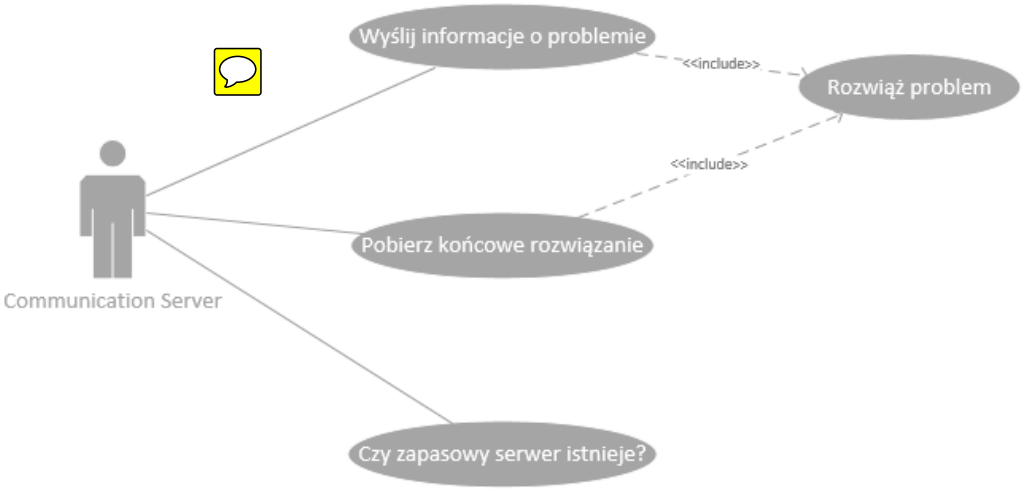
3.2 Instalacja Pluginów

Komponenty klastra odpowiedzialne za obliczenia tj. *Task Manager* i *Computational Node* przy każdym odpytaniu serwera o listę klas problemów, które mogą rozwiązać będą przeszukiwać folder `TaskSolvers` znajdujący się w katalogu z aplikacją, w celu poszukiwania bibliotek z klasami implementującymi interfejs `TaskSolver`. Gdy komponentowi zostanie zlecone zadanie, ten wykorzystując mechanizm refleksji wyszuka w katalogu odpowiednią klasę i stworzy jej instancję. Obiekt ten umożliwi realizacją zleconego zadania.
Instalacja nowych pluginów w komponentach systemu będzie sprowadzała się wyłącznie do wklejenie biblioteki z klasą implementującą interfejs `TaskSolver` do wyżej wymienionego katalogu.

Rozdział 4

Computational client

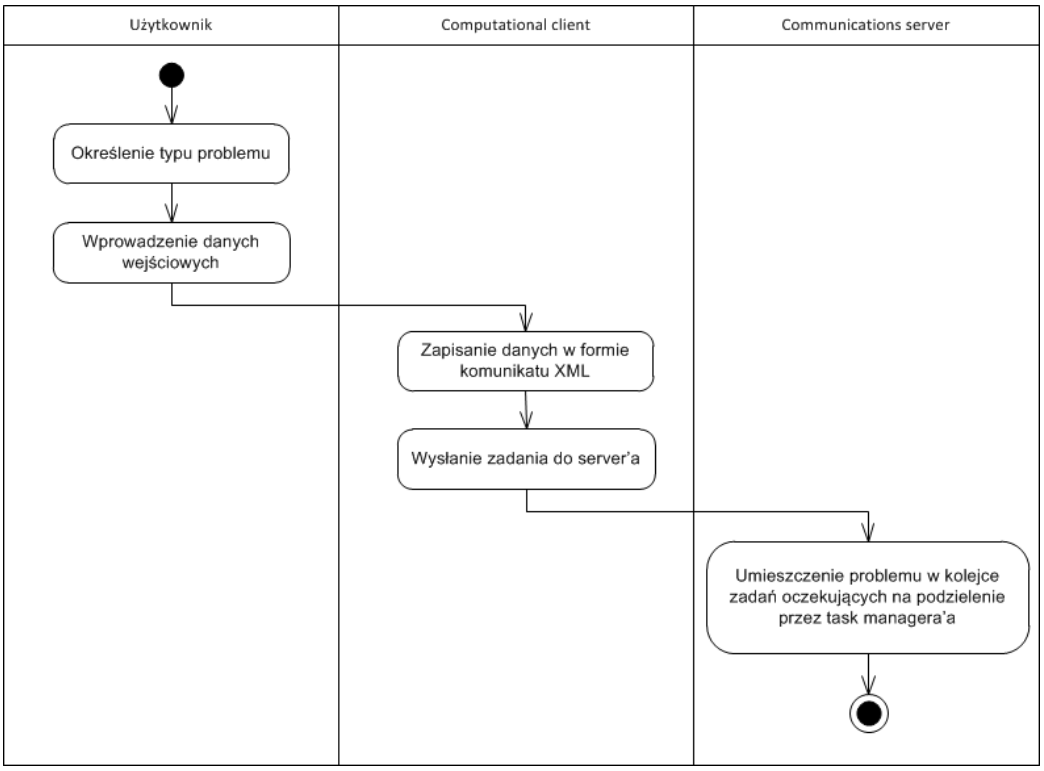
Zadaniem *computational client'a* jest pobranie danych wejściowych od użytkownika i wysłanie do *communications server'a* zlecenia rozwiązania podanego problemu. Gdy obliczenia zostaną zakończone, klient odbiera od serwera rozwiązanie zadania i prezentuje otrzymane wyniki użytkownikowi. Aby sprawdzić czy konkretne zadanie zostało już obliczone, *computational client* pobiera z *server'a* listę wszystkich ukończonych zleceń.



Rysunek 4.1: Diagram aktywności Serwer - Klient

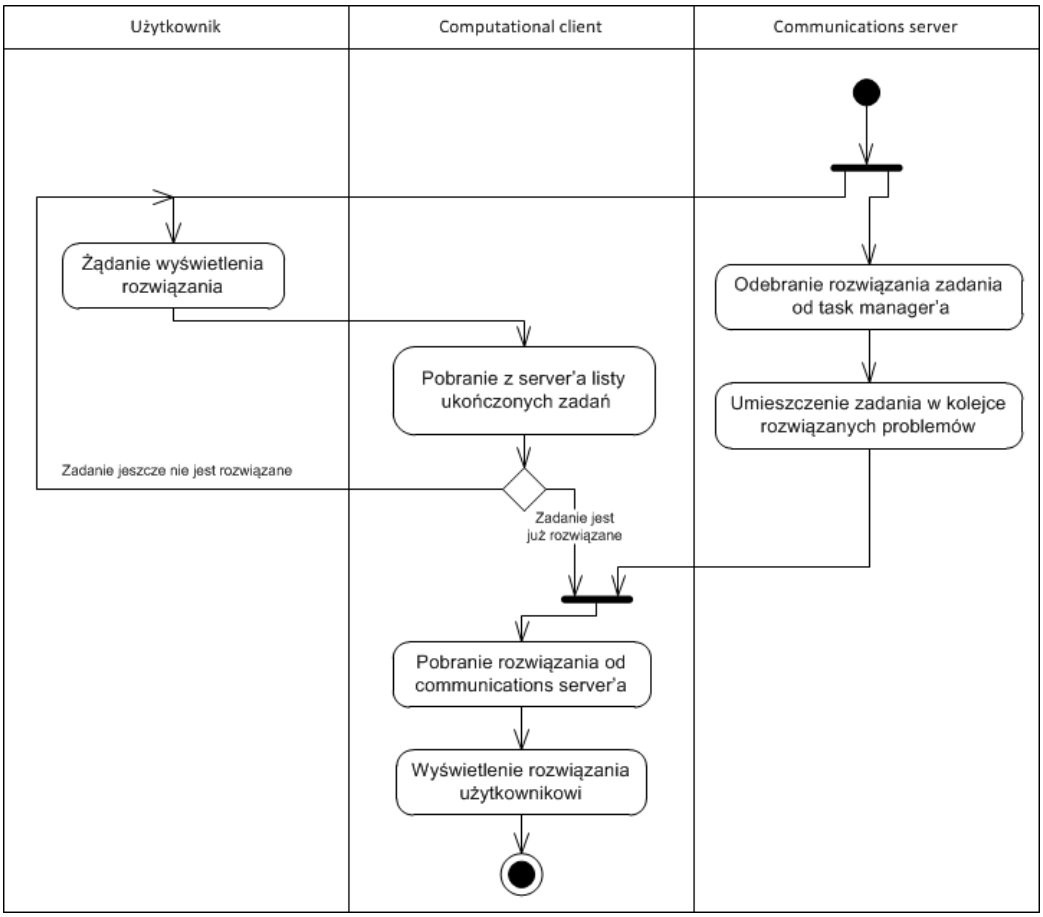
4.1 Zlecenie rozwiązania problemu

Użytkownik zleca klastrowi obliczeniowemu rozwiązanie zadania za pośrednictwem *computational client'a*. Określa typ problemu, jaki ma zostać obliczony oraz wprowadza dane wejściowe. *Computational client* zapisuje odebrane dane w formacie XML, a następnie wysyła komunikat do *communications server'a*. Aplikacja kliencka nawiązuje połączenie z *server'em* używając adresu IP, zapisanego w pliku konfiguracyjnym. Po odebraniu zlecenia od *client'a*, *server* umieszcza nowe zadanie w kolejce problemów odpowiedniego typu oczekujących na rozwiązanie. Zadanie znajduje się w kolejce dopóki któryś z *task manager'ów* potrafiących obsłużyć problem tej klasy nie zakończy wcześniejszych obliczeń i nie będzie mógł się nim zająć.



4.2 Pobranie rozwiązania

Po otrzymaniu rozwiązania problemu od *task manager'a*, *communications server* umieszcza je na liście ukończonych zadań. Gdy użytkownik chce zobaczyć rozwiązanie swojego problemu, *computational client* wysyła do *server'a* zapytanie i otrzymuje listę wszystkich wyników. Jeśli poszukiwane zadanie znajduje się na tej liście to znaczy, że *computational cluster* skończył już jego obliczanie. *Client* pobiera wtedy z *server'a* rozwiązanie i wyświetla je użytkownikowi. Jeżeli obliczenia nie zostały jeszcze zakończone, użytkownik dostaje komunikat, że nadal trwa rozwiązywanie zadania.



Rozdział 5

Communication server

Communication server sprawuje najważniejszą rolę w działaniu całego systemu. Zajmuje się przyjmowaniem zadań oraz przydzielaniem zasobów potrzebnych do ich rozwiązania, dba o poprawne i bezpieczne przechowywanie wyników i innych obiektów komunikacyjnych. Pełni rolę swojego rodzaju pośrednika w komunikacji między innymi elementami klastra.

Communication server ze względu na swoje znaczenie ma możliwość podłączenia dodatkowego communication server'a uruchomienia w trybie *backup*, z którym będzie prowadzona dodatkowa synchronizacja danych, w celu bezproblemowego działania w przypadku utraty jednego z nich.

Dla zwiększenia czytelności przedstawiony zostanie tryb normalny, różnice w trybie backup zostaną uwidocznione w osobnym podrozdziale.

5.1 Cykl życia serwera

Communication server zaraz po uruchomieniu sprawdza poprawność własnej konfiguracji, istnienia niezbędnych zależności (bazy danych). Gdy wszystko przebiegnie pomyślnie tworzy dwa wątki - nasłuchujący oraz operacyjny.

Wątek nasłuchujący zajmuje się oczekiwaniem i obsługiwaniem wstępnym nadchodzących komunikatów z innych komponentów klastra obliczeniowego. Konkretne zachowanie zależne jest od rodzaju komunikatu, jeżeli otrzymujemy zapytanie o parametry serwera odpowiedź finalna jest udzielana od razu, ale jeśli jest to zlecenie podzadania, poprawne przyjęcie jest sygnalizowane dopiero po synchronizacji z drugim communication server'em, jeśli taki jest ustawiony i dostępny.

Wątek operacyjny zaś, nieustannie (gdy ma taką możliwość) dba o redundancję danych wymieniając się danymi z innym communication server'em będącym w trybie *backup* oraz rozsyła komunikaty przydzielając poszczególne zadania, ze względu na zachowanie ciągłości synchronizacji ilość przydzielonych zadań w jednym cyklu jest dobierana eksperymentalnie i całkowicie konfigurowalna. Co jakiś czas wątek ten sprawdza również obecność wszystkich komponentów i aktualizuje ich listy obsługiwanych wtyczek.

Ogólny diagram aktywności communication server'a można znaleźć na rysunku 5.1.

5.2 Cykl życia zadania

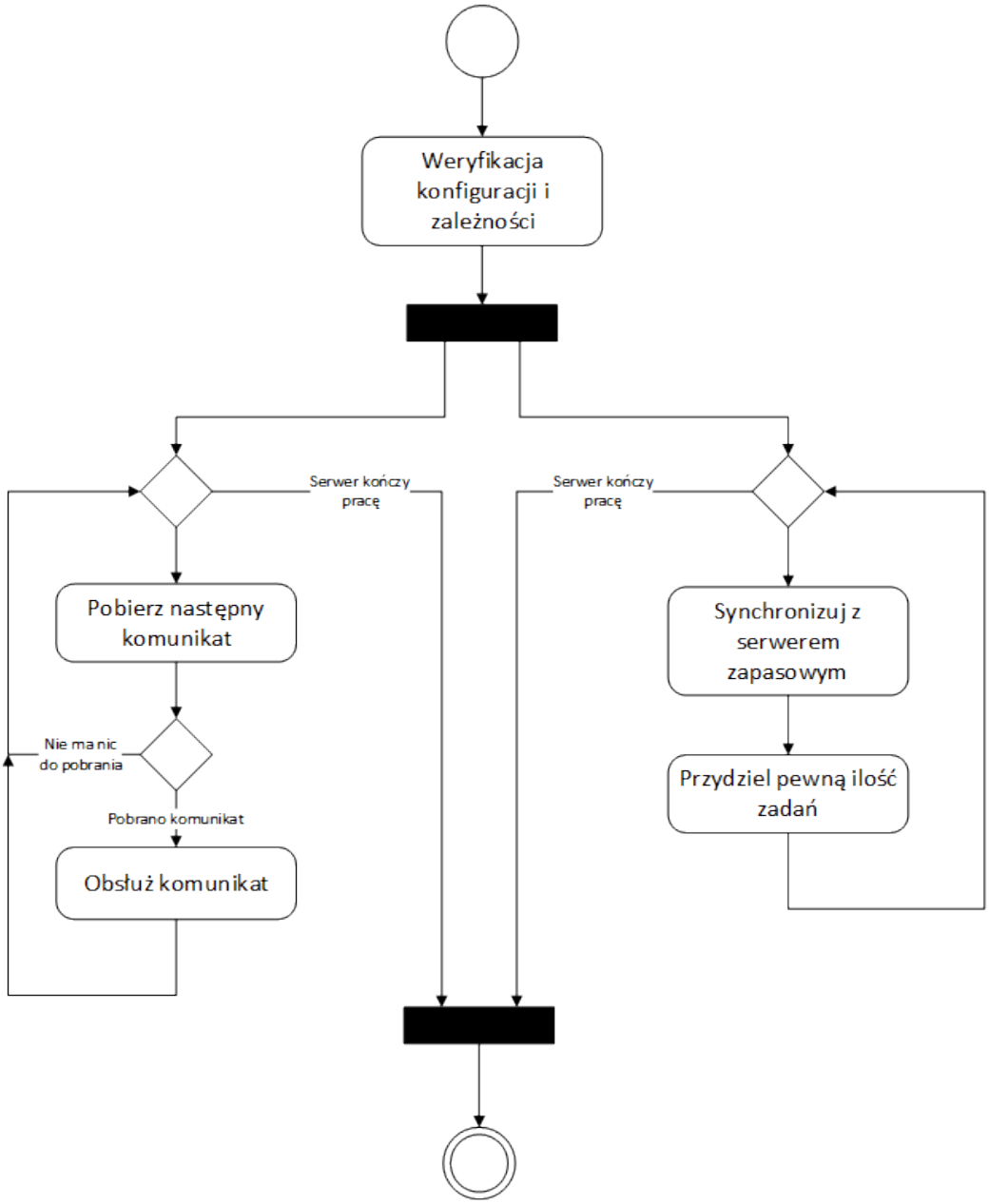
Dla każdego zadania możemy wyodrębnić kilka jego głównych stanów. Nowo zlecone zadanie trafia nieprzetworzone do odpowiedniej kolejki, oczekując na przesłanie do task manager'a i podzielenie na łatwiejsze do rozwiązania podproblemy. Po podzieleniu, zadanie czeka na rozwiązanie wygenerowanych podzadań, a po ich otrzymaniu wygenerowane jest rozwiązanie, które oczekuje potem na pobranie przez użytkownika; lub zgłaszane są dodatkowe podzadania potrzebne do wygenerowania poprawnego wyniku.

5.3 Przydzielanie zadań

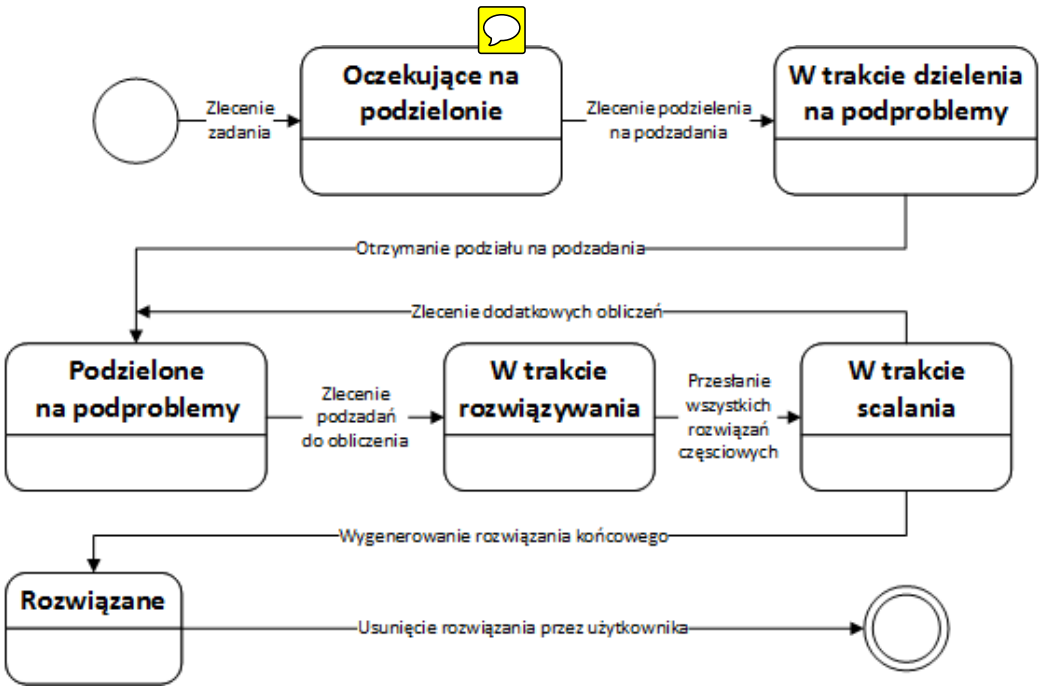
W celu optymalnego przechowywania zleconych przez użytkownika zadań opracowaliśmy strukturę ich przechowywania w której główny podział stanowi rodzaj problemu (TSP, DVRP itp.), a jego w obrębie znajdują się osobne listy FIFO dla nowo zgłoszonych problemów, podproblemów do rozwiązania oraz podproblemów do scalenia. Rodzaj problemu dla przyspieszenia obliczeń przechowywuje także informacje o menadżerach zadań i węzłach obliczeniowych, które potrafią go obsłużyć.

Communication server w momencie wyboru zadania filtruje rodzaje problemów w poszukiwaniu takich, które są mają oczekujące zadania oraz są rozwiązywalne, czyli posiadają do dyspozycji wolny task manager oraz mają zlecone jakieś zadanie. Spośród tych rodzajów wybieramy zadanie zlecone najwcześniej, by zachować oczekiwaną kolejność zajmowania się z zadaniami.

Przy przydzielaniu podzadań dla poszczególnych computational node'ów



Rysunek 5.1: Ogólny diagram aktywności communication server'a



Rysunek 5.2: Diagram stanów zadania

postępujemy analogicznie.

Rozwiązania podzadań odsyłamy do task manager'a w celu scalenia po otrzymaniu ich wszystkich. Menadżer po analizie odpowiedzi może zdecydować o wygenerowaniu rozwiązania końcowego lub zlecić podzadania dodatkowe i odłożyć końcową odpowiedź na termin późniejszy.

Nie należy zapominać również o zadaniach które zostały zlecone, ale z jakichś powodów nie zostały wykonane w oczekiwanym czasie. Takie zadania wracają do kolejki i czekają na ponowne przydzielenie.

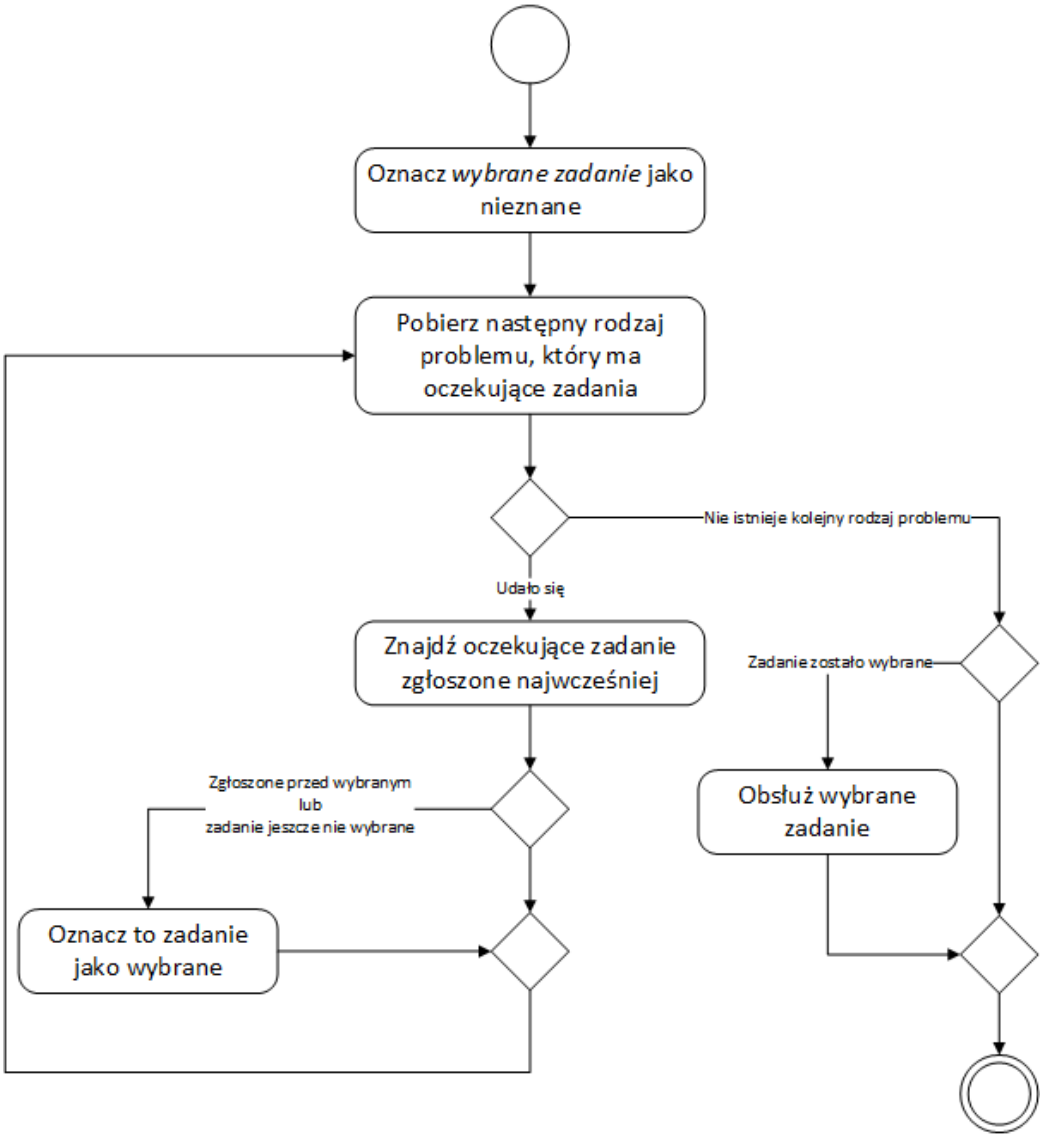
5.4 Tryb backup

Serwer w trybie backup zachowuje się bardzo podobnie do serwera w trybie normalnym. Najistotniejszą różnicą jest to, że stara się on być jedynie nośnikiem danych i nie wykonywać żadnych operacji na dostępnych zasobach. W przypadku próby zlecenia mu zadania podczas gdy nie jest on jedynym serwerem przekierowuje je do głównego communication server'a. Jeżeli zaś communication server nie odpowiada, przejmuje jego obowiązki. W momencie przywrócenia sprawności serwera głównego, synchronizują się one w nor-

malnym trybie, a serwer główny staje się w pełni operacyjny.

5.5 Synchronizacja między serwerami

Obiekty w czasie stworzenia i zapisania do jednej z baz danych oznaczane są jako obiekty do synchronizacji i są wymieniane między serwerami w następnym cyklu w formie wiadomości synchronizacyjnej. Podobnie dzieje się z zmodyfikowanymi obiektami (modyfikowane mogą być jedynie pola z datą przydzielenia pracy i informacją o zleceniobiorcy). Po otrzymaniu potwierdzenia odebrania są oznaczone jako zsynchronizowane. Jeśli potwierdzenie nie zostanie otrzymane, wykonywana jest retransmisja po odpowiednim czasie oczekiwania.



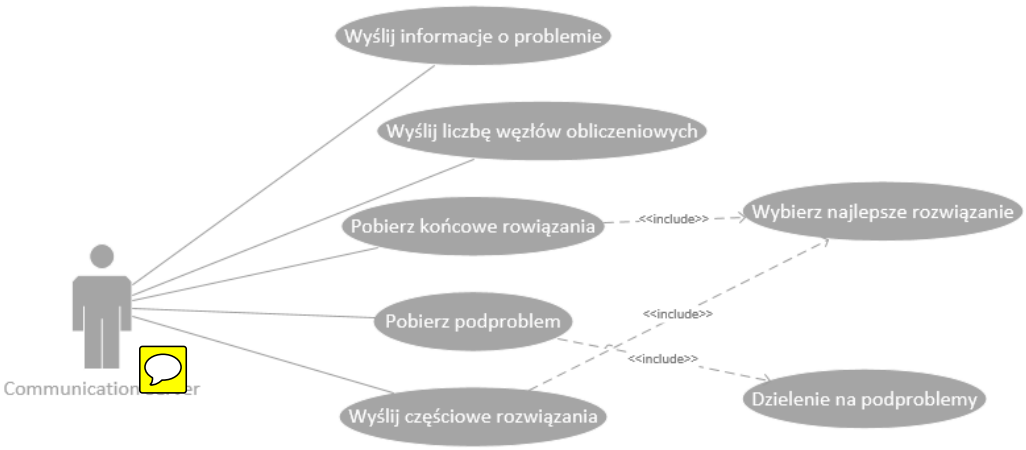
Rysunek 5.3: Wybór zadania przez communication server

Rozdział 6

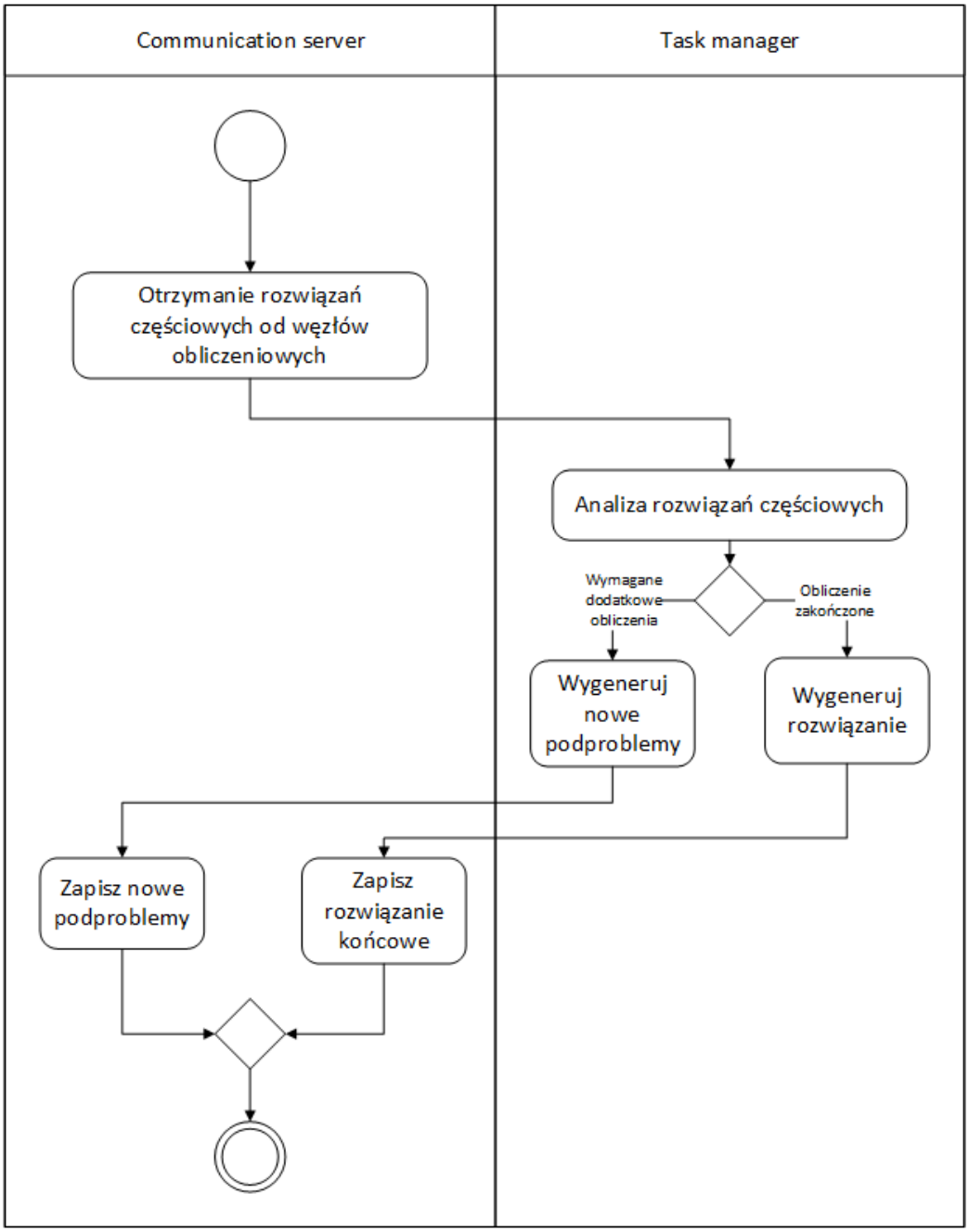
Task manager

Task Manager do podziału zadania wykorzystuje obiekt implementujący interfejs `TaskSolver`. Na tym etapie nie są obliczane konkretne drogi (w przypadku DVRP), a tylko ich identyfikatory. *Task Manager*, zamawia w Task Solverze podział zadania. Każdy Node dostaje potem polecenie obliczenia opcji między jednym a drugim identyfikatorem, pewien przedział problemów. Obliczanie konkretnych dróg nie miałoby sensu, oznaczałoby wykonanie przez Task Managera pracy Node'a.

Częściowe rozwiązania są przechowywane przez *Communications Server*. Gdy wszystkie zlecone zadania zostaną rozwiązanie *Task Manager* otrzymuje je wszystkie w celu utworzenia końcowego rozwiązania. Łączenie zadania polega na wyborze najbardziej optymalnego rozwiązania. Odbywa się to znowu przy pomocy odpowiedniej klasy implementującej interfejs `TaskSolver`, wykorzystując metodę klasy problemu, która scala rozwiązania.



Rysunek 6.1: Diagram aktywności CS - TM

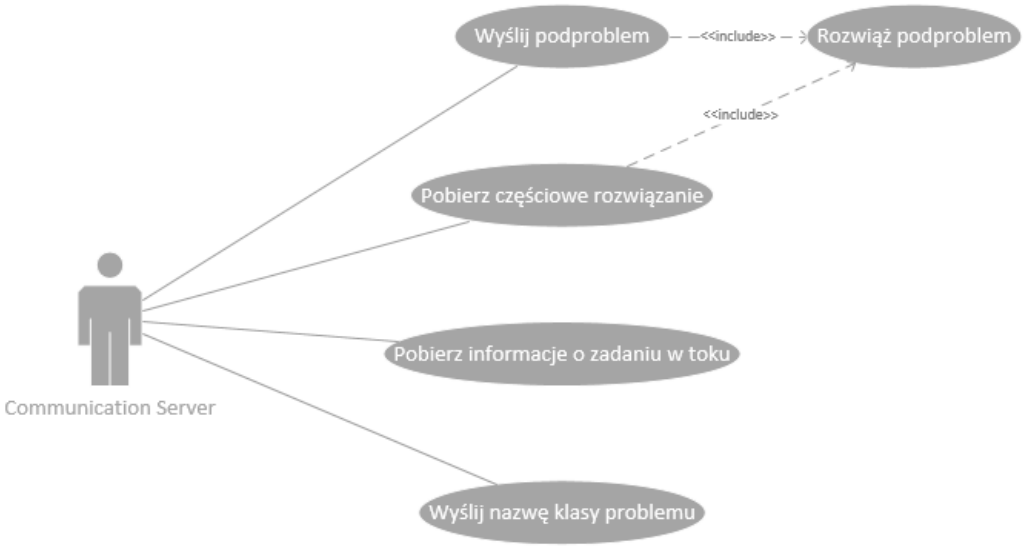


Rysunek 6.2: Diagram aktywności dla próby scalenia rozwiązania.

Rozdział 7

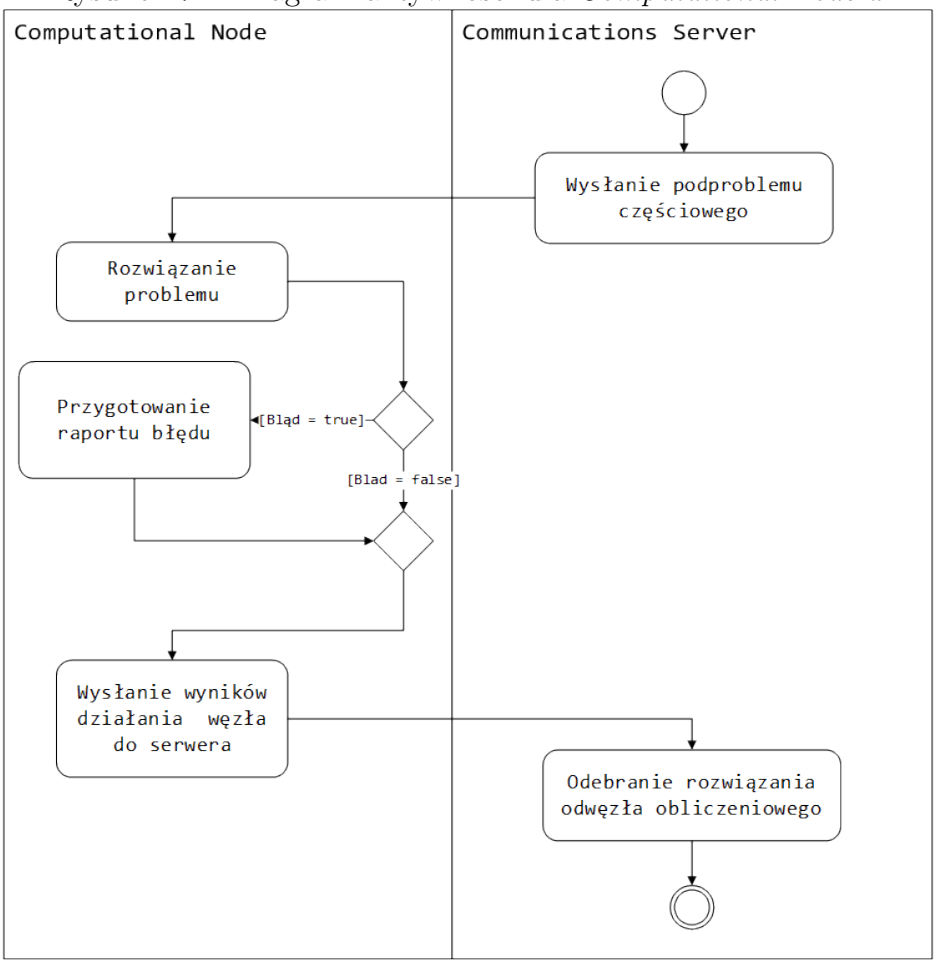
Computational node

Zadanie *Computational node'a* sprowadza się do stworzenia obiektu odpowiedniej klasy implementującej interfejs `TaskSolver` oraz uruchomienia w nim odpowiedniej metody. Podczas pracy *Computational Node'a* utrzymywana jest komunikacja pomiędzy węzłem a serwerem. Po zakończeniu obliczeń węzeł wysyła do serwera odpowiedni komunikat z rozwiązaniem zadania.



Rysunek 7.1: Diagram aktywności CS - CN

Rysunek 7.2: Diagram aktywności dla *Computational Node'a*



Rozdział 8

Komunikacja

8.1 Nawiązywania połączenia

Uruchomienie całego systemu rozpoczyna się od uruchomienia *Communications Server'a*. Od tej chwili komponenty systemu tj. *Computational node'y* i *Task Managery* mogą zgłaszać swoją obecność w klastrze. Każdy z komponentów systemu posiada w swoim pliku konfiguracyjnym (w węzłach `MainServerAddress` i `BackupServerAddress`) **adres IP** i **port** *Communications Server'a* oraz serwera zapasowego. Zaraz po uruchomieniu każdy komponent zgłasza swoją obecność do głównego serwera. Po trzech nieudanych próbach nawiązania połączenia następuje wysłanie identycznej informacji do serwera backup'owego. Każdy z komponentów w takiej wiadomości informuje o swoim rodzaju podaje IP i port na którym działa. Na tej podstawie serwer będzie komunikował się z tymi elementami systemu.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ConnectionMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <ComponentType>TaskManager</ComponentType>
4   <IP>0.0.0.0</IP>
5   <Port>1111</Port>
6 </ConnectionMessage>
```

Listing 8.1: Wiadomość wysyłana przez komponent włączający się do systemu

Parametr `ComponentType` informuje o rodzaju komponentu systemu. Wartości, które może przyjąć ten parametr to:

- `TaskManager` - w przypadku gdy wiadomość pochodzi od *Task Manager'a*,
- `ComputationalNode` - gdy wiadomość pochodzi od *Computational Node'a*

Zadaniem *Communications Server'a* jest utrzymywanie listy aktywnych komponentów systemu, oraz przechowywanie informacji na temat klas problemów, które dane komponenty obsługują. W tym celu serwer regularnie, co pewien określony czas będzie odpytywał wszystkie swoje komponenty prosząc o listę klas problemów możliwych do rozwiązania. W przypadku gdy takiej odpowiedzi nie otrzyma uznaje komponent za wyłączony i usuwa jego dane z pamięci. W przypadku otrzymania odpowiedzi na żądanie, serwer na podstawie otrzymanych danych uzupełnia/aktualizuje informacje o rozwiązywalnych problemach przez komponenty.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <SolvableProblemListMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <ProblemClassList>
4     <ProblemClass problemClassName="name1" problemClassId="ee28fec2
          -6361-4a58-aaf1-b9ff0f509743"/>
5     <ProblemClass problemClassName="name2" problemClassId="f9ed0a8f-a9f1
          -494a-aea6-68ffc533934e"/>
6   </ProblemClassList>
7 </SolvableProblemListMessage>
```

Listing 8.2: Odpowiedź komponentu na żądanie serwera

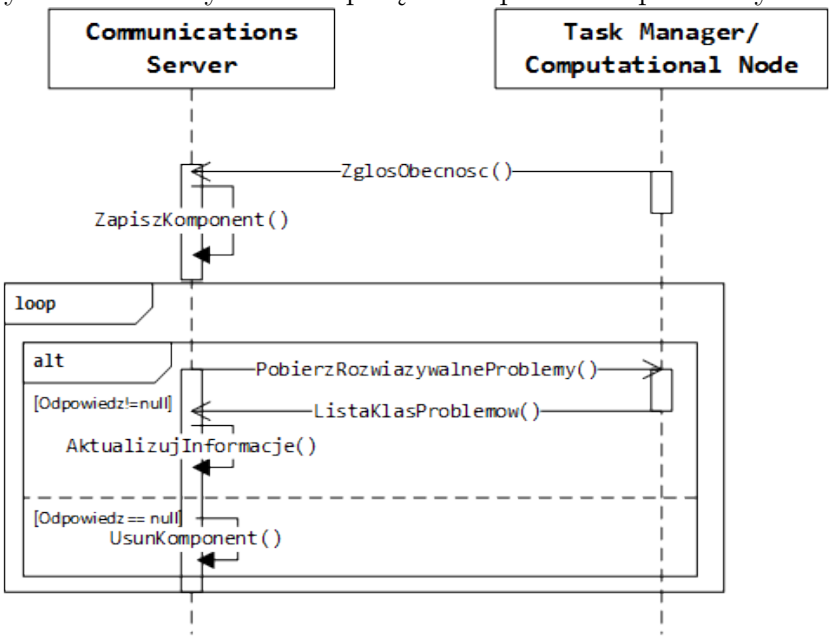
W wiadomości, dla każdego typu problemu zostają przekazane dwie informacje:

- `problemClassName` - nazwa problemu,
- `problemClassId` - guid problemu, który w jednoznaczny sposób identyfikuje typ problemu.

8.2 Lista rozwiązywalnych problemów

Proces rozwiązywania zadania przez *Computational Cluster* rozpoczyna się na poziomie *Computational client'a*. Przed wysłaniem problemu do rozwią-

Rysunek 8.1: Uzyskiwanie połączenia przez komponent systemu



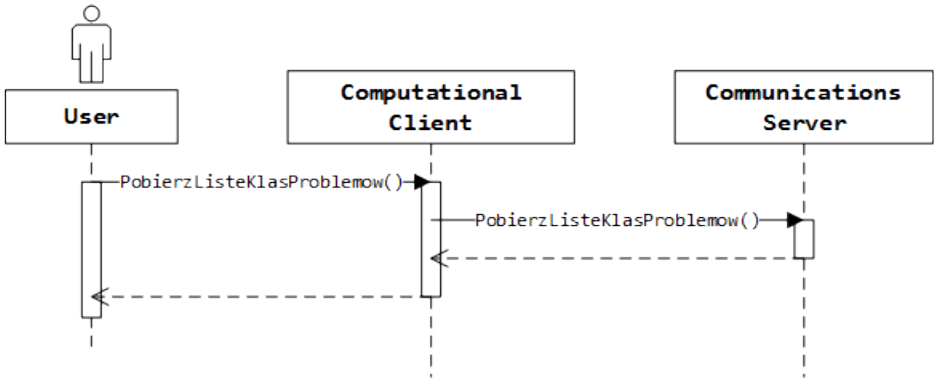
zania, aplikacja kliencka musi pobrać z serwera informację na temat klas problemów rozwiązywalnych w danej chwili przez *Computational Cluster* (wynika ona z obecnie dostępnych *Menadżerów zadań* i *Computational Node'ów*). Aplikacja kliencka odpytuje *serwer komunikacyjny* o potrzebne informacje. Serwer w odpowiedzi na to żądanie odsyła wiadomość, w której zawarte są informacje o wszystkich typach problemów rozwiązywalnych przez klaster. Informacje zostają przetworzone i przedstawione użytkownikowi. Wiadomość przesłana do *Computational Clienta* jest identyczna z tą, którą serwer otrzymuje od pozostałych komponentów systemu.

8.3 Rozwiązywanie zadania

Po wykonaniu czynności opisanych w poprzednim rozdziale, użytkownik może zlecić zadanie *klastrowi obliczeniowemu Computational Client* wysyła do serwera wiadomość z informacją o typie rozwiązywanego problemu i dane wejściowe zadania.

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

Rysunek 8.2: Diagram sekwencji pobierania informacji z serwera przez *Computational Client'a*




```
<TaskOrderMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <problemClassId>4fbcbba1-014e-4643-b60f-f7888a95bb54</problemClassId>
  <Data>dane w postaci XML</Data>
</TaskOrderMessage>
```

Listing 8.3: Zlecenie rozwiązania zadania przez *Computational Client'a*

Wiadomość zawiera 2 parametry konieczne do stworzenia i rozwiązania zadania:

- `problemClassId` - identyfikator klasy problemu, umożliwiający zidentyfikowanie, które części systemu są w stanie rozwiązać dany problem
- `Data` - parametry typu `String` zawierający dane wejściowe dla danego typu problemu. Dane te przedstawione są w formacie XML odpowiednim dla danego typu problemu.

Serwer po otrzymaniu wiadomości generuje specjalny identyfikator, który przypisuje do zadania. Będzie on wykorzystany do identyfikacji poszczególnych podzadań oraz umożliwi klientowi zidentyfikowanie zleconego zadania.

```
<?xml version="1.0" encoding="UTF-8"?>
<TaskTokenMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <TaskId>4fbcbba1-014e-4643-b60f-f7888a95bb54</TaskId>
</TaskTokenMessage>
```

Listing 8.4: Informacja z tokenem zwracana do aplikacji klienckiej

Taki token zostaje również odesłany do aplikacji klienckiej, która zapisze go w swoim pliku konfiguracyjnym. Poniżej przykład prezentujący odpowiedni wpis:

```
<TaskList>
  <Task taskname="customTaskName1" taskToken="ee28fec2-6361-4a58-
      aaf1-b9ff0f509743"/>
  <Task taskname="customTaskName2" taskToken="4fbcbba1-014e-4643-
      b60f-f7888a95bb54"/>
</TaskList>
```

Węzeł `Task` odpowiada jednemu zleconemu zadaniu przez użytkownika. Każdy taki węzeł posiada dwa atrybuty:

- `taskToken` - to identyfikator zadania nadany przez klaster obliczeniowy, dzięki niemu użytkownik będzie miał możliwość pobrania rozwiązania zadania,
- `taskname` - nazwa zadania nadana przez użytkownika, która umożliwi mu zidentyfikowanie zadania, nazwa obecna wyłącznie na poziomie aplikacji klienckiej.

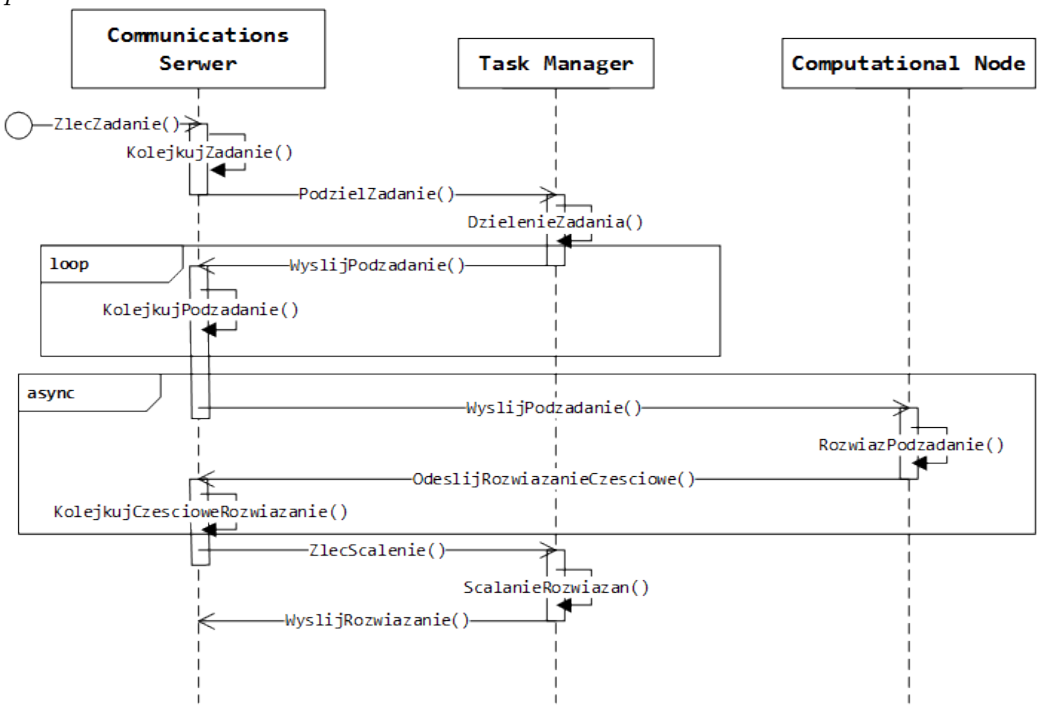
Computational Cluster wykonuje ciąg czynności koniecznych do otrzymania rozwiązania, tzn:

(1) przesłanie polecenia podziału zadania od *serwera* do *Task Manager'a* - wiadomość w formacie otrzymanym od *Computational Client'a*,

(2) podział zadania i odesłanie stworzonych podproblemów prowadzących do otrzymania rozwiązania. *Task Manager* dokonuje podziału zadania, każdemu z nich przydzielając unikalny identyfikator. Wiadomość zwrotna zawiera informacje o typie rozwiązywalnego problemu, identyfikatora zadania, identyfikatora podzadania, oraz danych wejściowych potrzebnych do rozwiązania zadania.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <SubtaskOrderMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <problemClassId>4fbcbba1-014e-4643-b60f-f7888a95bb54</
        problemClassId>
4   <TaskId>d104742a-76bb-4942-92bb-9f92149f4863</TaskId>
5   <SubtaskId>d104742a-76bb-4942-92bb-9f92149f4863</SubtaskId>
```

Rysunek 8.3: Diagram sekwencji pobierania informacji z serwera przez *Computational Client'a*




```
6   <Data>dane podzadania XML</Data>
7 </TaskResultMessage>
```

Listing 8.5: Zlecenie podzadania

(3) przesłanie każdego z podzadań do *węzłów obliczeniowych*

(4) przeprowadzenie obliczeń i odesłanie częściowych rozwiązań do *serwera*. Odsyłana wiadomość zawiera informacje o typie rozwiązywalnego podproblemu, identyfikatorze zadania, identyfikatorze podzadania oraz rozwiązaniu częściowym problemu.

(5) przesłanie częsciowych rozwiązań do *Task Managera* w celu stworzenia rozwiązania. Przesłanie wiadomości wcześniej zebranych od *Computational Node'ów*.

(6) odesłanie pełnego rozwiązania zadania do *serwera* w identycznym formacie jak te, które serwer odsyła do *Computational Clienta*

Zadanie przesłane do podziału przez *Communications server* jest w formacie otrzymanym od *Computational Client'a*

8.4 Odczytanie rozwiązania

Proces odczytywania rozwiązania zadania rozpoczyna się od załadowania listy zadań zleconych przez daną aplikację z pliku konfiguracyjnego do którego wcześniej zostały wpisane identyfikatory poszczególnych zadań. Użytkownik wybiera zadanie którego rozwiązanie chce pobrać. Aplikacja odpytuje serwer o rozwiązanie, w wyniku czego otrzymuje informację na jego temat. Wiadomość, którą wysyła aplikacja kliencka w węźle `TaskId` zawiera identyfikator zadania którego rozwiązania oczekujemy.

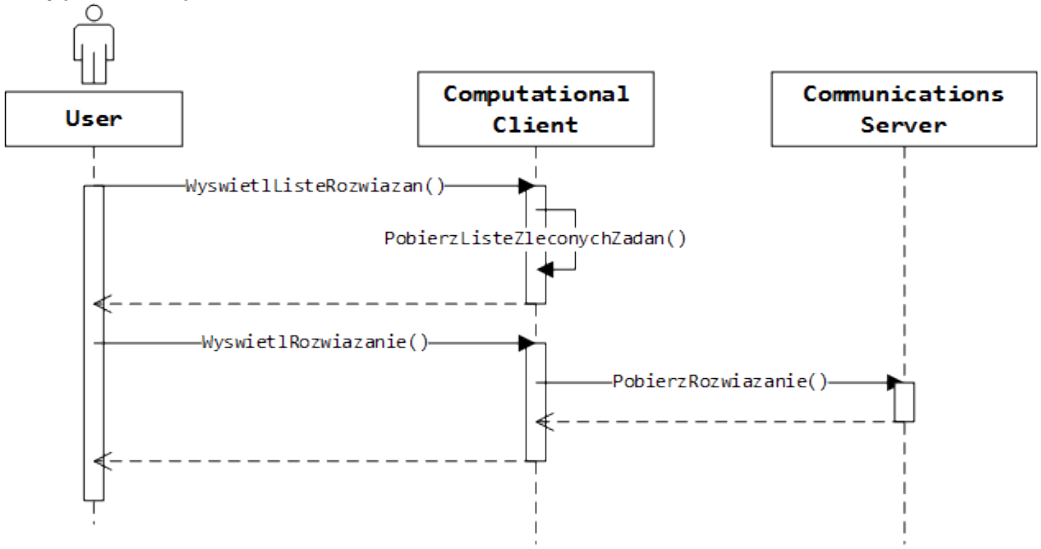
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <TaskResultMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <problemClassId>4fbcbba1-014e-4643-b60f-f7888a95bb54</problemClassId>
4   <TaskId>d104742a-76bb-4942-92bb-9f92149f4863</TaskId>
5   <Status>Done</Status>
6   <Data>rozwiazaniew postacie XML</Data>
7 </TaskResultMessage>
```

Listing 8.6: Rozwiązanie zadania

Wiadomość otrzymana od serwera składa się z 4 węzłów:

- `problemClassId` - identyfikator klasy problemów, której dotyczy zadanie. Informacja potrzebna do odpowiedniego wyboru pluginu, który odpowiednio sparsuje i przedstawi wyniki użytkownikowi.
- `TaskId` - identyfikator zadania
- `Status` - dwie możliwe wartości to `Done` - jeżeli zadanie zostało ukończone oraz `InProgress` - jeżeli rozwiązywanie zadania nadal trwa.
- `Data` - w przypadku gdy status przyjmuję wartość `Done` pole zawiera rozwiązanie zadania w postaci XML, w przeciwnym przypadku wartość jest pusta

Rysunek 8.4: Diagram sekwencji pobierania informacji z serwera przez aplikację kliencką



8.5 Schema

8.5.1 Uzyskiwanie połączenia przez komponenty

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema attributeFormDefault="unqualified" elementFormDefault="
       qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:simpleType name="components">
4      <xs:restriction base="xs:string">
5        <xs:enumeration value="TaskManager"/>
6        <xs:enumeration value="ComputationalNode"/>
7      </xs:restriction>
8    </xs:simpleType>
9
10   <xs:simpleType name="ip">
11     <xs:restriction base="xs:string">
12       <xs:pattern value="[0-9]{1,3}(\.[0-9]{1,3}){3}"/>
13     </xs:restriction>
14   </xs:simpleType>
15
16   <xs:element name="ConnectionMessage">
17     <xs:complexType>
18       <xs:sequence>
19         <xs:element type="components" name="ComponentType"/>
```

```
20        <xs:element type="ip" name="IP"/>
21        <xs:element type="xs:integer" name="Port"/>
22      </xs:sequence>
23    </xs:complexType>
24  </xs:element>
25 </xs:schema>
```

8.5.2 Lista rozwiązywalnych problemów przez klaster obliczeniowy

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
      qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="guid">
4     <xs:restriction base="xs:string">
5       <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
            ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
            -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
6     </xs:restriction>
7   </xs:simpleType>
8
9   <xs:element name="SolvableProblemListMessage">
10    <xs:complexType>
11      <xs:sequence >
12        <xs:element name="ProblemClassList" minOccurs="0">
13          <xs:complexType>
14            <xs:sequence>
15              <xs:element name="ProblemClass" maxOccurs="unbounded"
                    minOccurs="0">
16                <xs:complexType>
17                  <xs:simpleContent>
18                    <xs:extension base="xs:string">
19                      <xs:attribute type="xs:string" name="
                            problemClassName" use="required"/>
20                      <xs:attribute type="guid" name="problemClassId" use=
                            "required"/>
21                    </xs:extension>
22                  </xs:simpleContent>
23                </xs:complexType>
24              </xs:element>
25            </xs:sequence>
26          </xs:complexType>
27        </xs:element>
```

```
28        </xs:sequence >
29      </xs:complexType>
30    </xs:element>
31 </xs:schema>
```

8.5.3 Zlecenie rozwiązania zadania

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
      qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="guid">
4     <xs:restriction base="xs:string">
5       <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
            ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
            -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
6     </xs:restriction>
7   </xs:simpleType>
8
9   <xs:element name="TaskOrderMessage">
10    <xs:complexType>
11      <xs:sequence>
12        <xs:element type="guid" name="problemClassId"/>
13        <xs:element type="xs:string" name="Data"/>
14      </xs:sequence>
15    </xs:complexType>
16  </xs:element>
17 </xs:schema>
```

8.5.4 Token wysyłany do aplikacji klienckiej

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
      qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="guid">
4     <xs:restriction base="xs:string">
5       <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
            ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
            -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
6     </xs:restriction>
7   </xs:simpleType>
8
9   <xs:element name="TaskTokenMessage">
10    <xs:complexType>
11      <xs:sequence >
```

```
12        <xs:element type="guid" name="TaskId"/>
13      </xs:sequence >
14    </xs:complexType>
15  </xs:element>
16 </xs:schema>
```

8.5.5 Żądanie o przesłanie rozwiązania zadania

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="guid">
4     <xs:restriction base="xs:string">
5       <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
            ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
            -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
6     </xs:restriction>
7   </xs:simpleType>
8
9   <xs:element name="TaskResultRequestMessage">
10    <xs:complexType>
11      <xs:sequence >
12        <xs:element type="guid" name="TaskId"/>
13      </xs:sequence >
14    </xs:complexType>
15  </xs:element>
16 </xs:schema>
```

8.5.6 Rozwiązanie Zadania

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="guid">
4     <xs:restriction base="xs:string">
5       <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
            ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
            -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
6     </xs:restriction>
7   </xs:simpleType>
8
9   <xs:simpleType name="status">
10    <xs:restriction base="xs:string">
11      <xs:enumeration value="Done"/>
```

```
      <xs:enumeration value="InProgress"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="TaskResultMessage">
    <xs:complexType>
      <xs:sequence >
        <xs:element type="guid" name="problemClassId"/>
        <xs:element type="guid" name="TaskId"/>
        <xs:element type="status" name="Status"/>
        <xs:element type="xs:string" name="Data"/>
      </xs:sequence >
    </xs:complexType>
  </xs:element>
</xs:schema>
```

8.5.7 Zlecenie rozwiązania podzadania

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="guid">
    <xs:restriction base="xs:string">
      <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
          ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
          -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="SubtaskOrderMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="guid" name="problemClassId"/>
        <xs:element type="guid" name="TaskId"/>
        <xs:element type="guid" name="SubtaskId"/>
        <xs:element type="xs:string" name="Data"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

8.5.8 Przesłanie rozwiązania podzadania

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="guid">
    <xs:restriction base="xs:string">
      <xs:pattern value="([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F
          ]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|(\{[0-9a-fA-F]{8}-[0-9a-fA
          -F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\})"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="SubtaskResultMessage">
    <xs:complexType>
      <xs:sequence >
        <xs:element type="guid" name="problemClassId"/>
        <xs:element type="guid" name="TaskId"/>
        <xs:element type="guid" name="SubtaskId"/>
        <xs:element type="xs:string" name="Data"/>
      </xs:sequence >
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Rozdział 9

Przykład algorytmu

W tym miejscu działanie klastra zostanie omówione na przykładzie DVRP. Problem marszrutyzacji można rozpatrywać jako rozwinięcie problemu komiwojażera i chińskiego listonosza. Dostawca np. pizzy ma za zadanie rozwieść towar do klientów. Z punktu początkowego może wyruszać wiele pojazdów. Na podstawie wag krawędzi optymalizowany jest czas, odległość albo cena. Dynamic w nazwie problemu oznacza, że część klientów będzie dostępna dopiero o konkretnej godzinie.

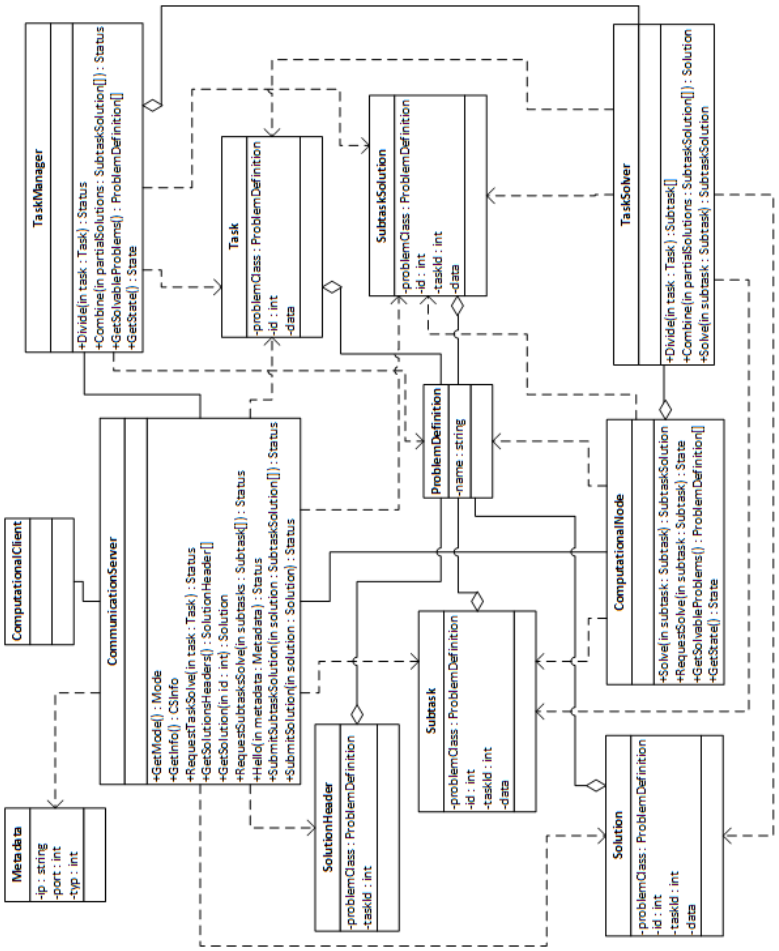
Klient wysyła do Serwera komunikacyjnego prośbę o rozwiązanie DVRP, listę krawędzi z wagami oraz liczbę dostawców. Serwer przekazuje problem do wolnego Menedżera zadań, który potrafi dzielić ten problem. Menedżer pyta serwer o liczbę dostępnych węzłów z Task solverem obsługującym DVRP.

Menedżer zadań, za pomocą Task Solvera używa metody klasy problemu, by wyznaczyć zakres identyfikatorów dróg dla poszczególnych Nodeów. Klasa problemu DVRP zawiera ściśle ustalony porządek możliwych podziałów na drogi i dostawców. Dlatego też Node będzie w stanie zrozumieć instrukcję, że ma obliczyć przykładowo możliwości między 5, a 19. W głównej pętli programu zostaną wstawione odpowiednie wartości.

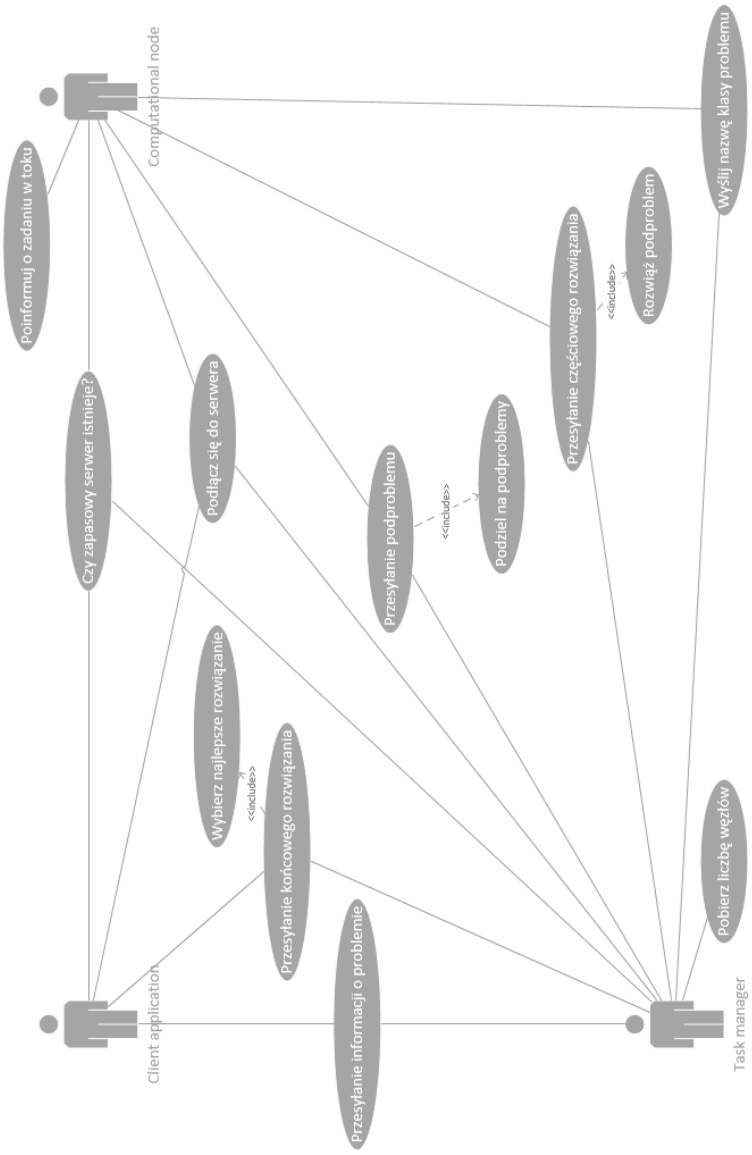
Węzeł za pomocą Task Solvera używa metody klasy problemu DVRP, by wyznaczyć przyznane mu trasy oraz ich koszty. Informuje też serwer, że jest w trakcie obliczeń. Gdy je zakończy, przesyła przez Serwer do Menadżera wszystkie pośrednie wyniki. Menadżer pod koniec obliczeń wybiera najbardziej optymalną trasę (najmniej kilometrów, najniższy koszt itp.), znowu za pomocą metody scalającej z klasy DVRP, odpalanej przez Task Solvera. Przesyła ją przez serwer do klienta.

Rozdział 10

Załączniki



Rysunek 10.1: Diagram klas



Rysunek 10.2: Diagram aktywnośći