Poznań University of Technology
Institute of Computing Science

Piotr Bazydło
Zachariusz Karwacki

Bluepath – biblioteka wspomagająca tworzenie programów rozproszonych

Praca magisterska

Promotor: dr inż. Jacek Kobusiński

Poznań, 2014

[tutaj wstaw kartę pracy]


Bluepath

Spis treści

Abstract

Bluepath – library supporting creating distributed applications

The aim of this work is to design a library, which, without complicated configuration or installation of multiple components, will allow programmers to implement distributed applications in easy way. A programmer, while writing an application using a parallel programming model based on threads, should be able to utilise the computing power of multiple machines in a cluster. The threads created by the programmer instead of being executed on one of machine's cores will be executed on one of the remote machines.

At the time of writing there wasn't available any popular, actively developed and commercially supported environment for creating distributed applications in .NET Framework. This kind of enviroment would allow programmers not only to avoid the need of solving many problems found in distributed programming by themselves but also to save time. One of the main goals was to make the library as modular as possible – this way Bluepath wouldn't strongly depend on the implementation of its components.

Rozdział 1
Wstęp

Programming is intrinsically very difficult.

— E. Dijkstra

Tworzenie aplikacji działających w klastrze jest złożone i wiąże się z nim wiele problemów takich jak wykrycie zakończenia czy dystrybucja danych pomiędzy maszynami. Dużo prostsze jest tworzenie aplikacji wykorzystujących wiele procesorów i rdzeni pojedynczej maszyny. W niniejszej pracy zaproponowano rozwiązanie czerpiące z obu podejść, pozwalające zastosować mechanizmy znane z programowania równoległego przy tworzeniu aplikacji rozproszonych.

W tym rozdziale przedstawione zostaną założony cel oraz zakres pracy, jej struktura, motywacja oraz podział prac pomiędzy autorów.

1.1 Cel i zakres pracy

Celem pracy było zaprojektowanie biblioteki, która, bez skomplikowanej konfiguracji czy instalowania wielu składników, pozwoli jej użytkownikom, programistom, w prosty sposób zaimplementować program przetwarzający dane w sposób rozproszony. Programista pisząc program w sposób podobny do tego w jaki pisałby program równoległy, powinien uzyskać program wykorzystujący moc obliczeniową wielu węzłów w klastrze. W takim przypadku wątki, zamiast wykonywać się na wielu rdzeniach jednego procesora, mogą wykonywać się na zdalnych maszynach.

1.2 Struktura pracy

W niniejszej pracy omówiono koncepcję oraz implementację biblioteki o kodowej nazwie Bluepath. Bieżący rozdział opisuje w dalszej części motywacje oraz wkład członków zespołu. Rozdział drugi stanowi wstęp teoretyczny obejmujący definicje oraz przegląd istniejących rozwiązań – środowisk przetwarzania rozproszonego.

W rozdziale trzecim została szczegółowo przedstawiona koncepcja i projekt systemu, a rozdział czwarty opisuje implementacje poszczególnych komponentów. Przykładowe zastosowania biblioteki przy implementowaniu aplikacji obliczeniowych przedstawia rozdział piąty, a wyniki testów jakościowych, wydajnościowych i ich analizę – rozdział szósty. W rodziale siódmym podsumowano przebieg realizacji, opisano napotkane problemy a także zaproponowano obszary, w których możnaby kontynuować prace nad usprawnieniem biblioteki.

1.3 Motywacja

W czasie, kiedy rozpoczynano pracę nie było dostępne popularne, aktywnie rozwijane i wspierane komercyjnie środowisko do przetwarzania rozproszonego dla .NET Framework (por. punkt 2.3). Tego typu środowisko pozwoliłoby użytkownikom uniknąć konieczności samodzielnego rozwiązywania wielu problemów przetwarzania rozproszonego oraz zaoszczędzić czas poprzez wykorzystanie dostarczonych mechanizmów.

1.4 Udział w pracy poszczególnych członków zespołu

Niniejsza praca została stworzona przez dwie osoby. Podział prac został przedstawiony poniżej.

Piotr Bazydło

- koncepcja rozwiązania,
- przegląd i analiza mechanizmów pamięci współdzielonej,
- przykładowe aplikacje – DLINQ, system uzupełniania wyrazów,
- implementacja części testów jakościowych i wydajnościowych,
- przygotowanie środowiska do przeprowadzenia testów wydajnościowych,
- implementacja zamków rozproszonych,
- implementacja planisty szeregującego zadania w oparciu o obciążenie węzłów,
- implementacja planisty szeregującego zadania wykorzystującego algorytm cykliczny,
- implementacja pamięci rozproszonej na bazie istniejącego rozwiązania,
- implementacja rozproszonych struktur danych i obiektów.

Zachariusz Karwacki

- przegląd istniejących rozwiązań,
- koncepcja rozwiązania,
- implementacja logiki wątku rozproszonego,
- implementacja zdalnego wykonawcy,
- implementacja lokalnego wykonawcy,
- implementacja mechanizmów logowania zdarzeń,
- implementacja usługi odnajdywania węzłów,
- implementacja zarządcy połączeń,
- implementacja interfejsu do komunikacji z systemem,
- implementacja części testów jakościowych i wydajnościowych,
- przykładowe aplikacje – MapReduce, obliczanie liczby PI,
- stworzenie skryptów dystrybuujących aplikację w klastrze.

Rozdział 2

Podstawy teorytyczne

W tym rozdziale przedstawione zostaną terminy wykorzystywane w dalszej części pracy oraz zestawienie rozwiązań podobnych tematyką do tej pracy.

2.1 Przetwarzanie równoległe

Jednym z modeli przetwarzania danych jest przetwarzanie równoległe. Polega ono na przetwarzaniu zbioru danych jednocześnie na wielu procesorach i rdzeniach. W ten sposób fragmenty przetwarzania, które nie są od siebie zależne mogą zostać wykonane w tym samym czasie – pozwala to skrócić czas przetwarzania.

Jednym ze sposobów prowadzenia przetwarzania równoległego jest wykorzystanie wątków. Wątki, czasami nazywane lekkimi procesami, wykonują kod niezależnie od siebie, ale działają we wspólnej przestrzeni adresowej, co pozwala współdzielić dane, ale wymaga zastosowania mechanizmów synchronizacji. Podstawowym mechanizmem synchronizacji są zamki. Służą one do realizacji sekcji krytycznych – fragmentów kodu, które mogą być wykonywane w danym momenie tylko przez jeden wątek.

2.2 Przetwarzanie rozproszone

Przetwarzanie rozproszone [**?**] jest szeroko stosowanym podejściem w przypadku przetwarzania dużej ilości danych, konieczności obsługi wielu zleceń użytkowników w jednostce czasu czy zapewnienia niezawodności systemu. Problemy, z jakimi zmagają się programiści aplikacji rozproszonych dotyczą m. in. obsługi awarii (wraz z rosnącą liczbą elementów systemu rośnie prawdopodobieństwo awarii jednego z nich), bezpieczeństwa (zapewnienia poufności, integralności i autentyczności komunikatów) i komunikacji. Trzeba brać pod uwagę możliwość utraty pakietu, niedostarczenia potwierdzenia odbioru, maksymalizować wydajność i przepustowość poprzez minimalizowanie liczby komunikatów i opóźnień komunikacyjnych. Wykorzystywane w praktyce protokoły sieciowe rozwiązują także wiele innych problemów, jak np. za-

tory w sieci czy zaległe żądania (ang. *outstanding requests*). W celu przyspieszenia tworzenia systemów rozproszonych zaproponowano wprowadzenie abstrakcji realizujących niezawodną komunikację sieciową i wymianę danych bez bezpośredniego udziału programisty.

2.2.1 DSM

Model rozproszonej pamięci współdzielonej (ang. *distributed shared memory*, DSM) [**?**] implementowany jest na poziomie systemu operacyjnego. Zakłada on udostępnienie wspólnej przestrzeni adresowej dla wszystkich komputerów biorących udział w przetwarzaniu. Aplikacje odwołują się do pamięci w tradycyjny sposób. W przypadku wystąpienia błędu braku strony system operacyjny zajmuje się sprowadzeniem jej do lokalnej pamięci. Mechanizm DSM nie jest obecnie wykorzystywany przy konstruowaniu systemów rozproszonych.

2.2.2 RPC

Zdalne wywoływanie procedur (ang. *remote procedure call*, RPC) [**?**] jest realizowane na poziomie języka programowania. Jego celem jest sprawić, by wywołanie metody na zdalnym komputerze było dla programisty tak proste, jak wywołanie lokalnej metody. Zadaniem serwera jest zdefiniowanie listy metod, które udostępnia i które mogą być wywoływane przez klienta. Zdalne wywoływanie procedur składa się z biblioteki odpowiedzialnej za realizację komunikacji oraz korzystających z niej, generowanych automatycznie, namiastki klienta (ang. *client stub*) i namiastki serwera (ang. *server stub*). Namiastka klienta odpowiada za:

- utworzenie bufora na wiadomość,
- wypełnienie go danymi – umieszczenie w nim jakiegoś rodzaju wskaźnika/uchwytu do metody oraz parametry wywołania – w procesie serializacji (ang. *serialization*, *marshalling*),
- wysłanie wiadomości przy pomocy biblioteki realizującej proces komunikacji,
- dekodowanie otrzymanego rezultatu – proces ten jest nazywany deserializacją (ang. *deserialization*, *unmarshalling*).

Namiastka serwera jest symetryczna względem namiastki klienta i do jej zadań należą:

- dekodowanie otrzymanego zlecenia,
- faktyczne wywołanie wskazanej metody,
- przygotowanie wiadomości zawierającej wynik,
- zlecenie wysłania odpowiedzi do klienta.

Głównym problemem w modelu RPC są: serializacja złożonych struktur danych, obsługa wskaźników oraz obsługa współbieżnych żądań przez serwer.

2.2.3 Problem detekcji zakończenia

Istotnym problemem jest detekcja zakończenia przetwarzania w systemie rozproszonym. Konstrukcja obrazu globalnego stanu systemu w celu oceny stanu poszczególnych węzłów jest problemem trudnym. Istnieje szereg algorytmów, które go adresują. Wymagają one wprowadzenia pewnych założeń co do właściwości systemu, jak np. kanały FIFO w algorytmie detekcji zakończenia dla systemów asynchronicznych [**?**] czy drzewiasta topologia przetwarzania w algorytmie Dijkstry-Scholtena dla modelu dyfuzyjnego [**?**].

2.3 Istniejące rozwiązania

Poniżej przedstawiono wybrane środowiska umożliwiające tworzenie aplikacji do rozproszonych obliczeń. W tabeli 2.1 znajduje się ich porównanie.

2.3.1 PVM

PVM (ang. *Parallel Virtual Machine*) [**?**] powstał w 1989 r. w Oak Ridge National Laboratory, a jego dalszy rozwój odbywał się na University of Tennessee. W 1993 r. została wydana wersja 3.0. Jest to zintegrowany zestaw bibliotek i narzędzi, które służą do projektowania aplikacji rozproszonych działających w środowisku połączonych siecią komputerów heterogenicznych. Główne założenia systemu obejmowały:

- używanie puli maszyn określonej przez użytkownika do realizacji przetwarzania oraz możliwość dodawania i usuwania komputerów podczas pracy,
- brak pełnego abstrahowania od typów maszyn, tj. aplikacje, które wykorzystywały specyficzne własności określonych maszyn mogły zostać im przypisane,
- model przetwarzania oparty o zadania, które można utożsamiać z procesami w systemie Unix oraz jawne przesyłanie komunikatów pomiędzy zadaniami,
- wsparcie dla heterogenicznych środowisk – różnych typów komputerów, sieci i aplikacji, a także obsługa sytuacji, w których różne maszyny korzystają z różnych reprezentacji danych.

2.3.2 MPI

MPI (ang. *Message-Passing Intefrace*) [**?**, **?**] jest standardem opisującym interfejs biblioteki wspomagającej tworzenie systemów opartych na przesyłaniu wiadomości

(ang. *message-passing systems*). Standard MPI skupia się na modelu programowania opartym o przesyłanie wiadomości. Definiuje również współbieżne operacje wejścia/wyjścia, dynamiczne tworzenie procesów oraz operacje zbiorowe – np. synchronizacja procesów przy pomocy barier (ang. *barrier synchronization*). Jednymi z głównych założeń jakie postawili sobie twórcy standardu są:

- zapewnienie wydajnej i wiarygodnej komunikacji,
- umożliwienie stworzenia implementacji działających w środowisku heterogenicznym,
- wygodny w wykorzystaniu interfejs dla języka C oraz Fortran.

2.3.3 Dryad i DryadLINQ

System Dryad [**?**] został stworzony przez zespół naukowców z Microsoft Research. Jego celem było zapewnienie niezawodnego środowiska do obliczeń rozproszonych na dużych zbiorach danych. System ten miał pozwalać programiście pisać programy wykonywane w klastrze bez posiadania umiejętności programowania równoległego bądź rozproszonego.

Przetwarzanie było modelowane jako skierowany graf acykliczny, w którym wierzchołki reprezentowały sekwencyjne programy, a krawędzie przepływ danych jednokierunkowymi kanałami. System Dryad na podstawie tak zamodelowanego przetwarzania tworzył graf przetwarzania, wykonywał go i w razie potrzeby modyfikował.

Programy w wierzchołkach były zapisywane przy pomocy języka SCOPE (ang. *Structured Computations Optimized for Parallel Execution*) [**?**]. Język ten posiadał składnię podobną do języka SQL (and. *Structured Query Language*) [**?**].

W celu uproszczenia przetwarzania na bazie systemu Dryad powstało środowisko DryadLINQ [**?**]. Środowisko to dostarczało implementację LINQ (ang. *Language Integrated Query*), która wykorzystywała środowisko Dryad do rozproszonego przetwarzania kolekcji. Jednym z ograniczeń wprowadzonych przez DryadLINQ w stosunku do LINQ było założenie, że funkcje przetwarzające obiekty kolekcji nie będą modyfikowały zmiennych zdefiniowanych poza nimi – w przypadku wykonania takiej operacji twórcy nie definiowali zachowania systemu.

W roku 2011 Microsoft zaprzestał rozwijania Dryad w ramach *High Performance Comupting Pack* [**?**], skupiając się na dostosowaniu Apache Hadoop do pracy pod kontrolą Windows Server i Windows Azure.

2.3.4 Hadoop

Przetwarzanie w modelu mapowania i redukcji – MapReduce [**?**] – zostało pierwotnie zaproponowane przez firmę Google. Hadoop to jego implementacja wydana na otwartej licencji. Środowisko korzysta z *Hadoop Distributed File System* (HDFS) [**?**]

– rozproszonego systemu plików o wysokiej odporności na awarie węzłów. Udostępnia on dane w sposób strumieniowy oraz jest dostosowany do obsługi dużych zbiorów danych (tj. rozmiaru mierzonego w terabajtach) i dużych plików. Twórcy środowiska wyszli z założenia, że „zmiana miejsca obliczeń jest mniej kosztowna niż przesłanie danych", w związku z tym HDFS udostępnia aplikacjom interfejs, który pozwala zmienić miejsce wykonania tak, aby odbyło się tam, gdzie fizycznie znajdują się dane.

2.3.5 Spark

Środowisko obliczeniowe Spark [**?**] powstało w wyniku prac prowadzonych na University of California w Berkeley. Od 2013 r. jako Apache Spark jest rozwijane przez Apache Software Foundation. Środowisko to może być użyte do przetwarzania w modelu nieograniczonym do dwóch faz jak w MapReduce, co upodabnia je bardziej do Dryad. Zastosowane w środowisku optymalizacje pozwoliły znacząco poprawić czas opóźnienia rozpoczęcia przetwarzania zadania w klastrze składającym się z tysięcy rdzeni [**?**]. Warto wspomnieć w tym miejscu o ciekawej abstrakcji pamięci udostępnianej przez środowisko Spark o nazwie *Resilient Distributed Dataset* (RDD) zapewniającej odtwarzanie danych utraconych w wyniku awarii.

2.3.6 Erlang

Język programowania Erlang [**?**] został zaprojektowany do tworzenia odpornych na awarie systemów rozproszonych. Jego historia sięga roku 1985, kiedy to w firmie Ericsson postanowiono „zrobić coś ze sposobem, w jaki tworzą aplikacje". W roku 2000 Erlang został upubliczniony na otwartej licencji. Podstawowe jego cechy to:

- izolacja procesów, brak współdzielonych struktur danych w pamięci operacyjnej, zamków, semaforów,
- procesy komunikują się przesyłając asynchroniczne wiadomości, które zawierają faktyczne dane, a nie referencje na zdalne obiekty,
- zdolność do wykrycia awarii i replikacja – procesy otrzymują sygnały w przypadku awarii obserwowanych procesów i muszą posiadać tyle danych, by przejąć zadania straconego węzła i kontynuować przetwarzanie,
- system udostępnia metodę określania przyczyn awarii procesu.

Z uwagi na mały narzut pamięciowy procesów (nazywanych także *aktorami*) programista może tworzyć ich bardzo wiele.

2.3.7 Project Orleans

W modelu programowania Orleans [**?**] zaproponowano wprowadzenie abstrakcji *wirtualnych aktorów*. W porównaniu do języka Erlang, środowisko wykonwacze działa tu na wyższym poziomie abstrakcji, uwalniając programistę od problemów takich jak obsługa awarii, odtwarzanie aktorów i zarządzanie rozproszonymi zasobami – w szczególności rozłożeniem aktorów na poszczególne węzły. W systemie Orleans przyjęto następujące założenia:

- aktorzy istnieją permanentnie (wirtualnie), programista nie tworzy ani nie niszczy samodzielnie reprezentujących ich obiektów,
- instancje aktorów (tzw. *aktywacje*) są tworzone automatycznie w momencie wysłania żądania do aktora, który nie istnieje lub gdy aktor jest typu *stateless worker*, tj. nie jest wymagane, aby istaniała tylko jedna instancja aktora danego typu (w przeciwieństwie do *single activation*),
- przezroczystość położenia – instancja aktora może istnieć na różnych maszynach w różnych chwilach, może też nie istnieć w ogóle fizycznie, aplikacja nie zna lokalizacji aktora,
- automatyczne skalowanie – niezależne instancje aktorów typu *stateless worker* mogą być uruchamiane w wielu egzemplarzach.

2.3.8 Podsumowanie

Nie ulega wątpliwości, że projektując nowe systemy warto trzymać się rozwiązań, które są aktywnie rozwijane i wspierane. Istotnym czynnikiem decydującym o wyborze lub odrzuceniu środowiska może być język programowania. Pomimo tego, że język jest tylko narzędziem i dla programisty, który poznał ich kilka, rozpoznanie kolejnego i napisanie w nim aplikacji zazwyczaj nie stanowi problemu, okazuje się, że część osób posiada preferowany język programowania. Podczas pisania w nim czuje się komfortowo, wie jakich problemów może się spodziewać i jak je rozwiązać. W związku z tym w tabeli 2.1 zestawiającej omówione środowiska obok modelu programowania uwzględniono także języki programowania wspierane przez te środowiska oraz wskazano rok opublikowania pierwszego i najnowszego wydania.

2.4 Definicje

Poniżej zdefiniowano pojawiające się w pracy pojęcia oraz wyjaśniono istotne terminy technologiczne.

Tabela 2.1: Porównanie wybranych środowisk przetwarzania rozproszonego

Środowisko	Język programowania	Model programowania	Pierwsze[a]/najnowsze wydanie
PVM	C/C++/Fortran	przesyłanie komunikatów	1989 / 2009
MPI	C/C++/Fortran	przesyłanie komunikatów	1994 / 2014[b]
Dryad	SCOPE / .NET (DryadLINQ)	przepływ danych w DAG	2009 / 2011
Hadoop	Java/Scala, Python/C++ [**?**]	mapowanie i reducja	2009 / 2014
Spark	Java/Scala/Python	niesprecyzowany	2014 / 2014
Erlang	Erlang	przesyłanie komunikatów	2000 / 2014
Orleans	.NET (C#/F#/VB)	przesyłanie komunikatów	2014 / 2014

[a]Pierwsze publiczne stabilne wydanie

[b]Implementacja Open MPI

2.4.1 Nazewnictwo

W tym miejscu należy zdefiniować pojęcia *biblioteka Bluepath*, *użytkownik* oraz *kod użytkownika*, które mają specjalne znaczenie.

Definicja 2.4.1.
Biblioteka Bluepath (w skrócie: **biblioteka**) jest przedmiotem niniejszej pracy. To zestaw klas, które wspomagają twórców aplikacji rozproszonych.

Definicja 2.4.2.
Mianem **użytkownika** określany jest programista korzystający z funkcji udostępnianych przez bibliotekę Bluepath podczas tworzenia aplikacji.

Definicja 2.4.3.
Pod pojęciem **kodu użytkownika** rozumiany jest kod źródłowy pochodzący spoza biblioteki Bluepath – będący częścią aplikacji, która ją wykorzystuje.

Termin ten odnosi się najczęściej do fragmentów kodu, które są wykonywane w ramach rozproszonego wątku (def. 3.3.1).

2.4.2 Terminy technologiczne

W pracy pojawia się szereg terminów technologicznych – głównie akronimów – które, w celu uniknięcia nieporozumień, zdefiniowano poniżej.

XML

XML (ang. *Extensible Markup Language*) to uniwersalny język znaczników (zwanych tagami) służący do reprezentacji danych.

XSD

XSD (ang. *XML Schema Definition*) to język, który służy do opisywania w sposób formalny struktury elementów dokumentu XML.

WCF

WCF (ang, *Windows Communication Foundation*) [**?**] to środowisko przeznaczone do implementacji usług sieciowych.

Zapora ogniowa

Zapora ogniowa (ang. *firewall*) to oprogramowanie filtrujące ruch sieciowy w oparciu o zestaw reguł.

Zapory ogniowe mają również możliwość modyfikacji pakietów, czego przykładem może być mechanizm NAT (ang. *network address translation*) [**?**, **?**] służący do zamiany adresu źródłowego, docelowego oraz numerów portów w celu umożliwienia komunikacji z użyciem jednego publicznego adresu IP wielu komputerom znajdującym się w sieci lokalnej i korzystającym z prywatnych adresów IP.

Rozdział 3

Koncepcja i projekt systemu

Proponowany model programowania miał być jak najbardziej naturalny dla programistów zaznajomionych z zagadnieniami programowania równoległego (por. 2.1) i korzystać z koncepcji wątków, pamięci rozproszonej oraz zamków. Wprowadzone abstrakcje wysokiego poziomu miały na celu ukrycie przed użytkownikiem leżącego u podstaw programowania rozproszonego (por. 2.2) mechnizmu przesyłania komunikatów.

3.1 Komunikacja

Każdy z węzłów (def. 3.1.1) w klastrze (def. 3.1.2) może komunikować się poprzez sieć z każdym innym węzłem. Przyjęto następujące założenia co do protokołów komunikacyjnych:

1. protokół zastosowany w warstwie transportowej powinien zapewnić niezawodne dostarczenie wszystkich wiadomości,
2. protokół działający w warstwie aplikacji powinien oczekiwać odpowiedzi na wysyłane żądania.

Nie jest wymagane zachowanie kolejności dostarczania wiadomości – w szczególności z uwagi na to, że w kanale znajduje się w danym momencie co najwyżej jedna wiadomość. Dla uzyskania transparentności procesu rozpraszania obliczeń sam proces przesyłania komunikatów pomiędzy węzłami jest ukryty przed użytkownikiem.

Zakładamy również, że bezpieczeństwo klastra opiera się na tym, iż działa on w odseparowanej sieci, tj. bez bezpośredniego dostępu z sieci publicznej (internetu). W związku z tym zastosowane protokoły komunikacyjne nie muszą zapewniać mechanizmów uwierzytelniania i autoryzacji.

Definicja 3.1.1.
Mianem **węzła obliczeniowego** (w skrócie: **węzła**) określamy komputer, na którym działa usługa zaimplementowana z użyciem biblioteki Bluepath, której celem jest zlecanie bądź wykonanie zleconych zadań.

Klaster systemu Bluepath składa się z węzłów obliczeniowych połączonych siecią. Rysunek 3.1 przedstawia schemat architektury systemu.

Definicja 3.1.2.
Klaster to zbiór węzłów obliczeniowych, które są zarejestrowane we wspólnej *usłudze odnajdywania węzłów* (def. 3.2.1).

3.2 Usługa odnajdywania węzłów

Jednym z podstawowych założeń jest posiadanie przez wszystkie węzły w systemie lokalnej wiedzy o wszystkich innych węzłach obecnych w klastrze. Informacje dotyczące stanu poszczególnych węzłów są dostarczane przez *usługę odnajdywania węzłów*. Usługa ta powinna obsługiwać sytuację rejestracji oraz wyrejestrowania się węzłów. System z założenia abstrahuje od sposobu realizacji tej usługi (czy będzie to system scentralizowany, czy rozproszony np. oparty na algorytmie plotkującym). W związku z pełnioną rolą, *usługa odnajdywania węzłów* jest kluczowym elementem przy implementacji wykrywania awarii węzłów w klastrze. Nowe węzły muszą zarejestrować się w usłudze. Po rejestracji są one monitorowane dopóki nie opuszczą klastra i wyrejestrują się lub ulegną awarii. Niedostępne węzły – tzn. takie, które nie odpowiedziały na zapytanie w określonym czasie – są usuwane z klastra.

Definicja 3.2.1.
Usługa odnajdywania węzłów to system umożliwiający węzłom odnajdywanie się wzajemnie w sieci i łączenie w klaster obliczeniowy.

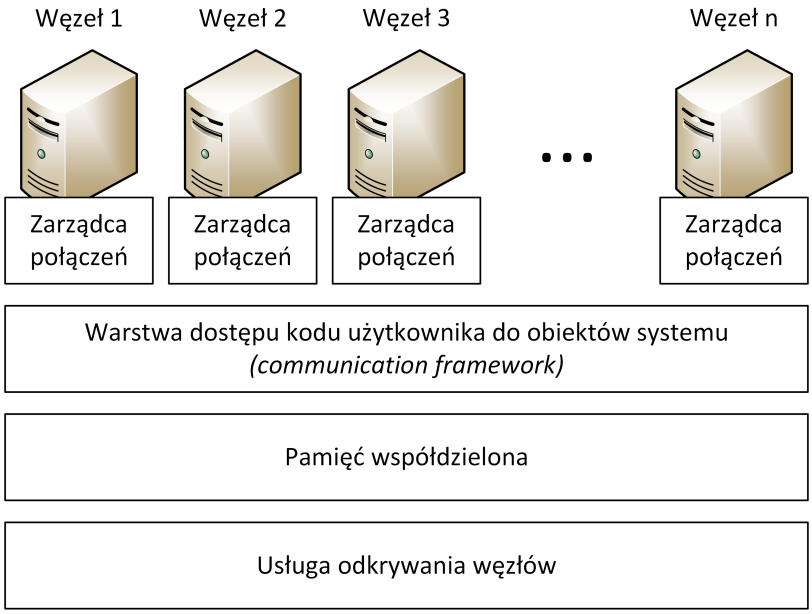
3.2.1 Zarządca połączeń

Każdy węzeł powinien przechowywać lokalnie obraz klastra, aby w momencie, kiedy zajdzie potrzeba zlecenia zadania jednemu ze *zdalnych wykonawców* nie musiał odpytywać *usługi odnajdywania węzłów*. Tym zadaniem zajmuje się *zarządca połączeń*. Przechowuje on też dodatkowe informacje, które mogą zostać wykorzystane przez *planistę*, takie jak statystyki obciążenia węzłów.

Definicja 3.2.2.
Zarządca połączeń to klient usługi odnajdywania węzłów, który odpowiada za rejestrowanie, wyrejestrowywanie i przechowywanie lokalnego obrazu stanu klastra na każdym z węzłów.

3.3 Wątek rozproszony

Specyfika systemu Bluepath wymaga rozróżnienia wątku systemu operacyjnego oraz *wątku rozproszonego* rozumianego jako jednostki przetwarzania w systemie. *Wątek rozproszony* jest tworzony na podstawie statycznej metody, która przyjmuje



Rysunek 3.1: Schemat architektury systemu

dowolną, liczbę parametrów i zwraca wartość. Wynik przechwytywany jest przez środowisko i udostępniany do odczytu na węźle, który zlecił wykonanie wątku. Rozproszony wątek może zostać wykonany na dowolnym węźle obliczeniowym oraz uruchamiać kolejne rozproszone wątki. Decyzja o wyborze miejsca wykonania wątku podejmowana jest z wykorzystaniem lokalnej wiedzy o stanie klastra przez *planistę*. Wątek rozproszony musi wykonać się w całości na jednym z węzłów – nie ma możliwości wstrzymania przetwarzania, przeniesienia wątku razem z aktualnym stanem i wznowienia obliczeń na innym węźle.

Definicja 3.3.1.

Wątek rozproszony reprezentuje jednostkę przetwarzania w systemie. Opakowuje fragment kodu, który może być wykonany na dowolnym węźle obliczeniowym.

3.3.1 Problem detekcji zakończenia

W punkcie 2.2.3 zasygnalizowano konieczność zaproponowania rozwiąznia problemu detekcji zakończenia. *Wątki rozproszone* tworzą drzewiastą strukturę przetwarzania – można ją określić mianem modelu dyfuzyjnego. Wyróżniamy tutaj inicjatora przetwarzania w korzeniu logicznego drzewa oraz jego następniki – procesy potomne, które ponadto mogą inicjować kolejne procesy. Użytkownik musi zapewnić, że proces-rodzic nie zakończy się, dopóki nie zakończą się jego procesy-dzieci korzystając z przeznaczonej do tego metody.

3.4 Wykonawca

W celu ograniczenia liczby zadań pełnionych przez *wątek rozproszony* zdefiniowano *wykonawcę* – element odpowiedzialny za wykonywanie wątków. Zastosowanie takiej abstrakcji pozwala potraktować przetwarzanie lokalne oraz zdalne w podobny sposób. Specjalizacje – *lokalny wykonawca* i *zdalny wykonawca* – wynikają z potrzeby realizacji innej logiki w przypadku wykonania kodu na lokalnej i zdalnej maszynie.

3.4.1 Lokalny wykonawca

Wszystkie wątki wykonywane lokalnie (zarówno te które pochodzą z lokalnego węzła jak i te zlecone przez inny węzeł) są zarządzane przez *lokalnych wykonawców*. Do zadań *lokalnego wykonawcy* należą:

- rozpoczęcie przetwarzania,
- przekierowanie parametrów,
- przechwycenie wyjątków,
- udostępnienie wyników wyższym warstwom po zakończeniu przetwarzania.

Definicja 3.4.1.
Lokalny wykonawca to obiekt, który odpowiada za faktyczne wykonanie otrzymanego zadania.

3.4.2 Zdalny wykonawca

Reprezentacją zdalnego uruchomienia *wątku rozproszonego* jest obiekt *zdalnego wykonawcy*. Zleca on wykonanie zadania węzłowi wybranemu przez *planistę* oraz monitoruje stan wykonania zadania (zakończenie, wystąpienie błędów). Jest on również odpowiedzialny za odebranie wyniku przetwarzania i udostępnienie go do odczytu na węźle zlecającym zadanie.

Definicja 3.4.2.
Zdalny wykonawca to obiekt, który odpowiada za zlecenie wykonania zadania na zdalnej maszynie.

3.5 Planista

Elementem systemu decydującym o wyborze miejsca wykonywania *wątku rozproszonego* jest *planista*. Każda jego realizacja musi być oparta na zdefiniowanym interfejsie. Dzięki temu użytkownik może przygotować własną implementację *planisty* dostosowaną do prowadzonego przetwarzania lub wybrać jedną z dostarczonych razem z biblioteką.

Definicja 3.5.1.
Planista odpowiada za wybór węzła, do którego wysłane zostanie zlecenie wykonania zadania. Realizuje zdefiniowaną strategię szeregowania zadań w klastrze.

3.6 Pamięć rozproszona

Często w ramach przetwarzania rozproszonego zachodzi potrzeba przesyłania danych między węzłami. W celu jej zrealizowania, jako część biblioteki *Bluepath*, dostarczana jest abstrakcja *pamięci rozproszonej*. Umożliwia ona synchronizację i wymianę danych między działającymi wątkami. Zaleca się jednak myślenie i korzystanie z niej jako pamięci masowej, ponieważ koszt dostępu do niej może być znacznie wyższy niż do pamięci operacyjnej. W szczególności nie należy jej mylić z DSM opisaną w punkcie 2.2.1.

Definicja 3.6.1.
Pamięć rozproszona to pamięć dostępna dla wszystkich węzłów, przez którą mogą wymieniać się danymi.

Implementacja pamięci rozproszonej nie była przedmiotem niniejszej pracy. Zdecydowano się zastosować jedno z istniejących rozwiązań spełniających następujące założenia:

- silna spójność danych – wszystkie węzły odczytują zawsze tę samą, najbardziej aktualną wartość,
- dostępność mechanizmów umożliwiających realizację operacji o semantyce atomowego porównania i zapisu (ang. *test and set*) lub atomowego odczytu i zapisu.

Warto zauważyć, że zgodnie z definicją 3.6.1 pod pojęciem *pamięci rozproszonej* będziemy rozumieć pamięć, która jest udostępniana przez co najmniej jeden węzeł i umożliwia współdzielenie danych między wszystkimi węzłami obliczeniowymi w klastrze.

W *pamięci rozproszonej* zastosowano klucze – unikalne ciągi znaków – jednoznacznie identyfikujące dane. W trakcie projektowania *pamięci rozproszonej* nie przewidziano zapewnienia hierarchii kluczy. Tego typu mechanizm, choć przydatny w pewnych przypadkach, może zostać dostarczony przez nadbudowane nad *pamięcią rozproszoną* przez użytkownika abstrakcje. Aby zapewnić wysoką użyteczność oraz możliwość rozbudowy *pamięci rozproszonej*, w podstawowym interfejsie przewidziane zostały następujące operacje:

- zapis (ang. *store*) – operacja zapisująca dane; w przypadku gdy podany klucz już istnieje operacja nie powiedzie się,

- zapis lub aktualizacja (ang. *store or update*) – operacja zapisująca dane; w przypadku gdy podany klucz istnieje, wartość zostanie nadpisana,
- aktualizacja (ang. *update*) – operacja aktualizująca dane – powiedzie się tylko wtedy, gdy podany klucz został wcześniej utworzony w *pamięci rozproszonej*,
- pobranie (ang. *retrieve*) – operacja pobierająca dane – powiedzie się tylko wtedy gdy podany klucz został wcześniej utworzony,
- usunięcie (ang. *remove*) – operacja usuwająca dane – powiedzie się tylko wtedy gdy podany klucz został wcześniej utworzony w *pamięci rozproszonej*.

Operacje te, choć wystarczające w większości zastosowań, zostały rozwinięte o odpowiedniki operujące jednocześnie na grupach kluczy – zostały one opisane w 3.6.2.

Dzięki tak zdefiniowanej semantyce operacji, na podstawie *pamięci rozproszonej* można również zbudować bardziej złożone mechanizmy, takie jak *zamki rozproszone*. Zamki są wbudowanym mechanizmem w proponowanej bibliotece i są dostępne poprzez interfejs *rozszerzonej pamięci rozproszonej*, który został opisany w punkcie 3.6.3.

Definicja 3.6.2.
Rozszerzona pamięć rozproszona to pamięć rozproszona (def. 3.6.1), która dodatkowo udostępnia abstrakcję zamków.

3.6.1 Rozproszone struktury danych i obiekty

Pamięć rozproszona – pomimo tego, że zapewnia operacje niezbędne do komunikacji w trakcie przetwarzania – może okazać się mechanizmem o zbyt niskim poziomie abstrakcji. W związku z tym biblioteka dostarcza struktury danych i obiekty oparte na *pamięci rozproszonej*: listę, słownik oraz licznik.

Lista

Głównym zadaniem *listy rozproszonej* jest przechowywanie i udostępnianie listy obiektów dostępnej w każdym *wątku rozproszonym*, który zna jej identyfikator. Lista ta zachowuje zgodność na poziomie interfejsu z listą dostarczaną przez środowisko .NET Framework. Podstawowe scenariusze wykorzystania *listy rozproszonej* nie powinny wymagać stosowania dodatkowych mechanizmów synchronizacji.

Słownik

Oprócz listy, często wykorzystywaną strukturą danych jest słownik. *Słownik rozproszony* pozwala użytkownikowi na współdzielenie między wątkami kolekcji typu klucz-wartość. Jednocześnie zapewnia poprawność wykonania współbieżnych operacji, np. atomowego dodania nowego klucza oraz zarejestrowania go w wewnętrznym indeksie kluczy. Podobnie jak lista rozproszona słownik identyfikowany jest przez

ciąg znaków, który musi być znany wszystkim wątkom chcącym uzyskać do niego dostęp.

Licznik

Jedną z podstawowych struktur rozproszonych jest *licznik rozproszony*. Oprócz standardowych operacji odczytu i ustawienia wartości, umożliwia on także atomowe pobranie i zwiększenie wartości.

Obiekt ten może być wykorzystany jako generator unikalnych kluczy lub do dynamicznego podziału kolekcji na fragmenty – każdy *wątek rozproszony* może pobierać wartość licznika i jednocześnie zwiększać jego wartość o n. W ten sposób rezerwuje sobie n obiektów począwszy od odczytanej wartości identyfikatora i każdy z wątków może operować na rozłącznym zbiorze obiektów.

3.6.2 Operacje zbiorcze

Głównym kosztem przetwarzania rozproszonego są narzuty komunikacyjne. Wykonywanie n żądań w celu pobrania n elementów z *pamięci rozproszonej* jest wysoce nieefektywne. W celu zminimalizowania kosztu komunikacji, dla każdej operacji na *pamięci rozproszonej* zdefiniowano jej odpowiednik przyjmujący jako parametr zbiór kluczy. Dodanie tego typu operacji pozwoli użytkownikowi w znaczący sposób zwiększyć efektywność swojego przetwarzania (np. poprzez jednorazowe pobranie do pamięci lokalnej przetwarzanych danych).

Definicja 3.6.3.
Operacje zbiorcze (ang. *bulk operations*) to operacje wykonywane jednocześnie na grupie obiektów o podobnych właściwościach znajdujących się w pamięci rozproszonej.

3.6.3 Zamki rozproszone

Przetwarzanie rozproszone oparte na *wątkach rozproszonych*, podobnie jak przetwarzanie równoległe, może wymagać zastosowania sekcji krytycznych. Operacje na *pamięci rozproszonej* zostały zdefiniowane w taki sposób, aby możliwe było stworzenie *zamków rozproszonych* – współdzielonych przez wszystkie węzły w klastrze (w odróżnieniu od zamków istniejących jedynie w obrębie jednego procesu bądź jednego komputera) – opartych o ten mechanizm. W związku z tym tworzenie i zarządzanie *zamkami rozproszonymi* jest częścią obowiązków *rozszerzonej pamięci rozproszonej*.

Ponieważ reszta biblioteki nie polega na sposobie implementacji zamków, a jedynie na gwarancjach jakie oferują, mogą one zostać zaimplementowane przez użytkownika w oparciu o mechanizmy niezwiązane z *pamięcią rozproszoną*.

Definicja 3.6.4.
Zamek rozproszony to zamek, który jest współdzielony przez wszystkie węzły w klastrze.

3.7 Logowanie zdarzeń

W celu umożliwienia przeglądu przebiegu przetwarzania system udostępnia mechanizmy do zapisu historii zdarzeń – zarówno tych wewnętrznych jak i zachodzących w kodzie użytkownika. Opcjonalnie, korzystając z abstrakcji *listy rozproszonej*, log jest zbierany w *pamięci rozproszonej*. Dzięki temu użytkownik nie musi samodzielnie zbierać logów z wszystkich węzłów w klastrze – wystarczy, że na jednym z nich zmaterializuje zawartość *listy rozproszonej* do pliku w celu dalszego przetwarzania i wizualizacji np. w narzędziach służących do eksploracji procesów.

3.8 Obsługa awarii

W obecnej wersji systemu nie przewiduje się mechanizmów odtwarzania i kontynuowania przetwarzania w przypadku awarii jednego lub większej liczby węzłów. Temat ten może być przedmiotem dalszych prac (por. punkt **??**).

Rozdział 4

Implementacja

Projekt systemu zakładał napisanie go w języku C# w środowisku .NET Framework 4.5 [?] i umożliwienie wykonywania kodu użytkownika z bibliotek skompilowanych do kodu zarządzalnego dla .NET Framework w wersji nie nowszej niż 4.5. Testy wykonania kodu użytkownika zostały przeprowadzone z użyciem aplikacji napisanych w językach C# oraz F#.

System Bluepath działa na klastrze węzłów obliczeniowych. Na każdym węźle musi zostać uruchomiona binarnie zgodna wersja aplikacji (w stopniu umożliwiającym wykonanie dowolnej metody z dowolnej klasy będącej częścią procesu za pomocą zserializowanego uchwytu otrzymanego w ramach zlecenia pod pewnymi warunkami opisanymi w punkcie **??**). W ramach jednego z wątków uruchamiana jest usługa nasłuchująca wywołań – zleceń, zapytań przychodzących od pozostałych węzłów.

4.1 Komunikacja

W sieci komunikacyjnej w warstwie transportowej działa protokół TCP [**?**, **?**, **?**], który realizuje gwarancje niezawodnego dostarczenia wiadomości z użyciem mechanizmu potwierdzeń i retransmisji zgodnie z założeniem przyjętym w punkcie 3.1 ppkt. 1. Komunikacja między węzłami została zrealizowana w modelu RPC (pkt. 2.2.2) za pomocą WCF (pkt. 2.4.2). Połączenia realizowane są za pomocą protokołu HTTP [**?**], a wiadomości są przesyłane zgodnie z protokołem SOAP [**?**, **?**, **?**], co spełnia wymagania postawione w punkcie 3.1 ppkt. 2. Z uwagi na losowy wybór numerów portów założono, że na maszynach nie działa zapora ogniowa (pkt. 2.4.2) blokująca połączenia przychodzące na portach o wysokich numerach. W sieci nie może także działać mechanizm NAT, co uniemożliwiłoby realizację bezpośredniej komunikacji między węzłami.

Wątek nasłuchujący jest reprezentowany przez obiekt klasy `BluepathListener`. Wewnętrznie tworzy on instancję klasy faktycznie nasłuchującej na losowym porcie na wiadomości zgodne z interfejsem `IRemoteExecutorService`, który został przedstawiony na listingu 4.1, a szczegółowy opis protokołu znajduje się w punkcie **??** poświęconym *wykonawcom*. Dokonywana jest też rejestracja w *usłudze odnajdywa-*

nia węzłów. Jeżeli w ramach jednego procesu zostanie utworzone kilka instancji klasy `BluepathListener` każda z nich będzie nasłuchiwała na innym porcie i posiadała własną kolekcję *lokalnych wykonawców*. Lista znanych *zdalnych wykonawców* jest współdzielona w ramach procesu przez wszystkie wątki nasłuchujące.

W systemie każdy węzeł może komunikować się z dowolnym innym, o ile zna jego adres IP i numer portu, pod którym działa usługa. Komunikaty, zarówno wywołania (ang. *request*) jak i wywołania zwrotne (ang. *callback*), mają charakter asynchroniczny. Istnieje również możliwość wyłączenia wywołań zwrotnych i przełączenia systemu w tryb pracy z odpytywaniem (ang. *polling*). Wprowadza on dodatkowe opóźnienia, co dyskwalifikuje go w przypadku prowadzenia w klastrze faktycznych obliczeń, znajduje jednak zastosowanie w niektórych scenariuszach realizowanych w środowisku testowym.

Listing 4.1: Interfejs IRemoteExecutorService

```
1  [ServiceContract]
2  public interface IRemoteExecutorService
3  {
4      [OperationContract]
5      Guid Initialize(byte[] methodHandle);
6
7      [OperationContract]
8      void Execute(Guid eId, object[] parameters, ServiceUri callbackUri);
9
10     [OperationContract]
11     void ExecuteCallback(Guid eid, RemoteExecutorServiceResult executeResult);
12
13     [OperationContract]
14     RemoteExecutorServiceResult TryJoin(Guid eId);
15
16     [OperationContract]
17     PerformanceStatistics GetPerformanceStatistics();
18 }
```

4.2 Usługa odnajdywania węzłów

Obecna implementacja zawiera usługę odnajdywania węzłów zrealizowaną w sposób scentralizowany, a każdy z węzłów podłączających się do klastra musi znać adres IP i numer portu, pod którym działa *usługa odnajdywania węzłów*. Usługa udostępnia metody służące do zarejestrowania się węzła (`Register`), wyrejestrowania się węzła (`Unregister`), pobrania listy zarejestrowanych w klastrze węzłów (`GetAvailableServices`), oraz pobrania statystyk wydajności wszystkich węzłów w klastrze (`GetPerformanceStatistics`).

Listing 4.2: Interfejs ICentralizedDiscoveryService

```
1  [ServiceContract]
2  public interface ICentralizedDiscoveryService
3  {
4      [OperationContract]
5      ServiceUri[] GetAvailableServices();
6
7      [OperationContract]
8      void Register(ServiceUri uri);
9
10     [OperationContract]
11     void Unregister(ServiceUri uri);
12
13     [OperationContract]
14     Task<Dictionary<ServiceUri, PerformanceStatistics>> GetPerformanceStatistics();
15 }
```

4.3 Wątek rozproszony

`DistributedThread` to klasa, której instancje reprezentują jednostkę przetwarzania w systemie czyli *wątek rozproszony* (patrz 3.3.1). Wątek taki tworzony jest na podstawie delegatu `Func` opakowującego statyczną (nazwaną lub anonimową) metodę. Liczba parametrów wejściowych takiej metody to maksymalnie 16 (jest to ograniczenie wprowadzone przez deklaracje typu `Func` dostępne w .NET Framework). Wszystkie typy parametrów oraz typ zwracany muszą być oznaczone atrybutem `Serializable`. Przykład tworzenia i uruchamiania przez użytkownika *wątku rozproszonego* z użyciem domyślnego *zarządcy połączeń* i *planisty* został przedstawiony na listingu **??**.

Środowisko zakłada izolację pomiędzy wątkami: *wątki rozproszone* działające w ramach jednego procesu-hosta nie mogą komunikować się ze sobą poprzez pamięć operacyjną, a parametry, z którymi wywoływana jest metoda są kopiowane do nowych instancji klas i struktur. Wyjątkiem jest tutaj dostarczana przez środowisko *pamięć rozproszona*. W tym przypadku użytkownik powinien stosować dostarczone wraz z nią mechanizmy synchronizacji.

Listing 4.3: Tworzenie i uruchamianie rozproszonego wÂątku

```
var thread = DistributedThread.Create(
    new Func<..., IBluepathCommunicationFramework, ...>((..., bluepath) => {
        ...
        return ...;
    }
));

thread.Start(...);
```

4.4 Wykonawca

Poniżej opisane zostały szczegóły protokołu komunikacyjnego (rys. ??) używanego przez węzły obliczeniowe oraz szczegóły implementacji *wykonawców*.

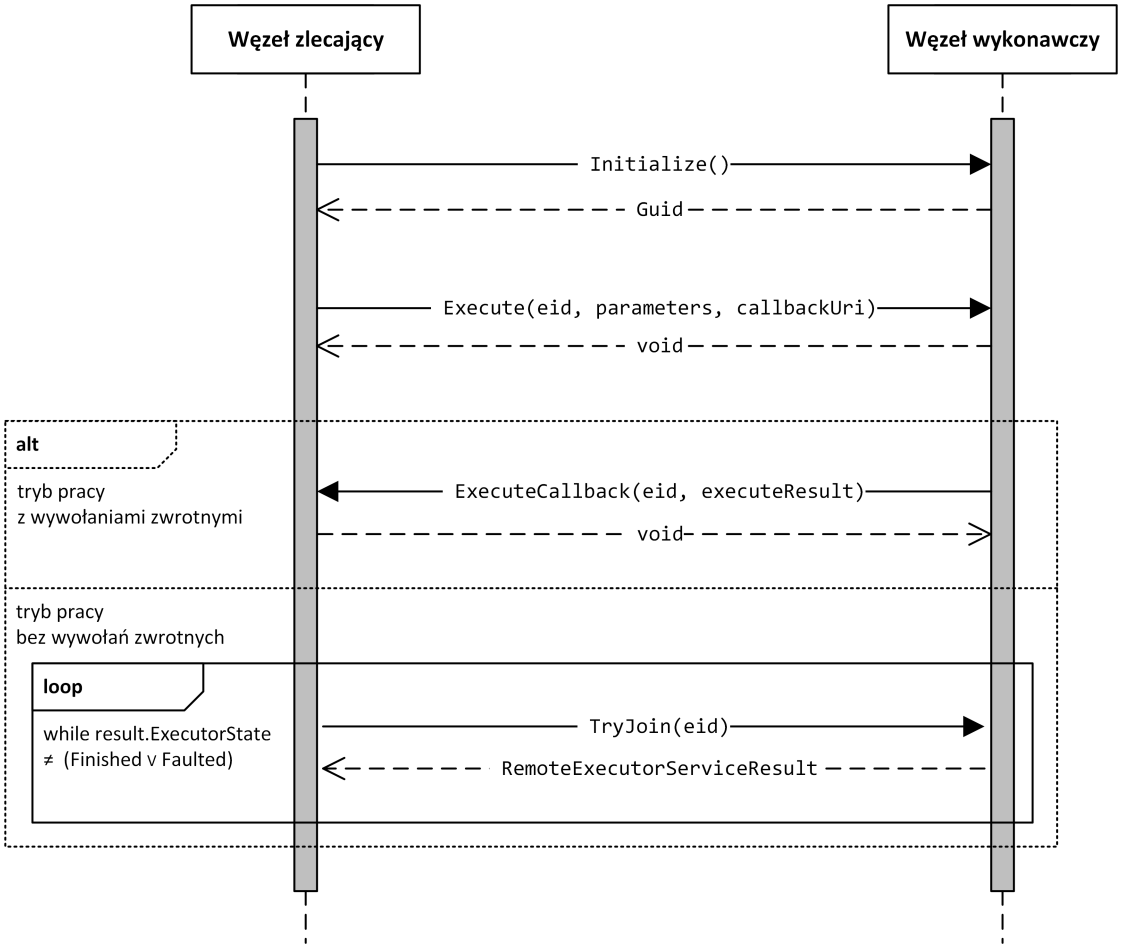
4.4.1 Inicjalizacja

Każdy wykonawca jest identyfikowany przez `eid` (ang. *Executor ID*) – unikalną 128-bitową liczbę (GUID, ang. *Globally Unique Identifier*), przy czym instancja *zdalnego wykonawcy* używa takiego samego identyfikatora jak *lokalny wykonawca*, który został zainicjowany do wykonania kodu użytkownika. Wiadomość inicjująca wątek `Initialize` zawiera element `methodHandle`, w którym przesyłany jest zserializowany binarnie i zapisany w kodowaniu Base64 uchwyt do metody. W odpowiedzi przesyłany jest identyfikator wykonawcy. Listing ?? przedstawia w skróconej formie przykładową przechwyconą kopertę SOAP z wiadomością `Initialize`.

Listing 4.4: Koperta SOAP – inicjalizacja zdalnego wykonawcy

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <Initialize xmlns="http://tempuri.org/">
      <methodHandle>AAEAAAD/////AQAAAAAAAAAMAgAAAD9[...]==</methodHandle>
    </Initialize>
  </s:Body>
</s:Envelope>
```

Listing ?? prezentuje fragment uchwytu do metody po zdekodowaniu i usunięciu znaków spoza drukowalnego zestawu symboli ASCII. DistributedPI to przykładowy program opisany szerzej w punkcie ??. Uchwyt dotyczy metody anonimowej (stąd wygenerowana przez kompilator nazwa `b__0`) zdefiniowanej w klasie `Bluepath.DistributedPI.Program`, która jako parametr przyjmuje `int` (`System.Int32`) a typem zwracanym jest `long` (`System.Int64`).



Rysunek 4.1: Diagram protokołu komunikacyjnego węzłów obliczeniowych

Listing 4.5: Zdekodowany fragment uchwytu do anonimowej metody

```
[...] System.Type[]
<RunTest>b__0 MBluepath.DistributedPI, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null Bluepath.DistributedPI.Program
Int64 <RunTest>b__0(Int32) (System.Int64 <RunTest>b__0(System.Int32)
System.UnitySerializationHolder Data  UnityType AssemblyName
```

4.4.2 Przesłanie parametrów, wykonanie

W kolejnej wiadomości, po inicjalizacji, węzeł zlecający przesyła parametry wywołania metody. Przechwycona koperta z komunikatem `Execute` znajduje się na listingu ??. Warto zauważyć, że ustawiona jest tutaj wartość pola `callbackUri`, co oznacza, że węzeł wywołujący otrzyma komunikat `ExecuteCallback` po zakończeniu przetwarzania po zdalnej stronie.

Listing 4.6: Koperta SOAP – przesÂłanie paramterĂłw

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <Execute xmlns="http://tempuri.org/">
      <eId>6a3ff0e2-601b-4683-a163-7f6da037c762</eId>
      <parameters xmlns:a="http://schemas.microsoft.com/2003/10/Serialization/
          Arrays" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:anyType i:type="b:int" xmlns:b="http://www.w3.org/2001/XMLSchema">
            10000</a:anyType>
      </parameters>
      <callbackUri xmlns:a="http://schemas.datacontract.org/2004/07/Bluepath.
          Services" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:Address>http://192.168.0.7:51000/BluepathExecutorService.svc</a:
            Address>
        <a:BindingType>BasicHttpBinding</a:BindingType>
      </callbackUri>
    </Execute>
  </s:Body>
</s:Envelope>
```

Lokalny wykonawca szereguje wątki do wykonania używając dostarczonej przez środowisko .NET Framework puli wątków (`ThreadPool`). Podejście to ma swoje wady – tracona jest kontrola nad tak stworzonymi wątkami, nie można ich przerwać lub sprawdzić ich stanu (uniemożliwia to np. realizację detekcji zakleszczenia). Biblioteka Bluepath zapewnia, że w przypadku, gdy w kodzie użytkownika wystąpi wyjątek, zostanie on przechwycony, zserializowany i udostępniony wywołującemu wątkowi do odczytu.

4.4.3 Wywołanie zwrotne

Po zakończeniu metody w trybie z wywołaniem zwrotnym, zdalna strona przesyła wynik, ew. wyjątki oraz czas jaki zajęło przetwarzanie do zlecającej maszyny. Przykładowy komunikat `ExecuteCallback` przedstawia listing **??**.

Listing 4.7: Koperta SOAP – wywoÂłanie zwrotne z wynikiem

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <ExecuteCallback xmlns="http://tempuri.org/">
      <eid>6a3ff0e2-601b-4683-a163-7f6da037c762</eid>
      <executeResult xmlns:a="http://schemas.datacontract.org/2004/07/Bluepath.
          Services" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:ElapsedTime>PT0.0125743S</a:ElapsedTime>
        <a:Error i:nil="true" xmlns:b="http://schemas.datacontract.org/2004/07/
            System"/>
        <a:ExecutorState>Finished</a:ExecutorState>
        <a:Result i:type="b:long" xmlns:b="http://www.w3.org/2001/XMLSchema">
            7857</a:Result>
      </executeResult>
    </ExecuteCallback>
  </s:Body>
</s:Envelope>
```

4.5 Planista

Wraz z systemem dostarczone zostały następujące typy *planistów*:

- `ThreadNumberScheduler` – szereguje zadania na najmniej obciążonym węźle pod względem liczby wykonywanych na nim wątków,
- `RoundRobinScheduler` – szereguje zadania korzystając z algorytmu cyklicznego (ang. *round robin*) [**?**].

Wszystkie implementacje *planisty* muszą implementować interfejs `IScheduler` dzięki czemu można zastosować implementację *planisty* dostosowaną do potrzeb przetwarzania i jest jednym z elementów modułowości systemu. Poniżej zostały szczegółowo opisane implementacje planistów dostarczonych z systemem.

4.5.1 Szeregowanie zadań w oparciu o obciążenie węzłów

Jednym z podstawowych sposobów szeregowania zadań jest próba równomiernego rozłożenia obciążenia węzłów. Zakładając, że wszystkie zadania mają podobny rozmiar, jako miarę obciążenia konkretnego węzła można przyjąć liczbę zadań, które są obecnie przetwarzane, lub oczekują na rozpoczęcie przetwarzania.

W przykładowej implementacji dostarczanej wraz z biblioteką *Bluepath* informacje o obecnym obciążeniu węzłów są okresowo odświerzane prz pomocy *usługi odnajdywania węzłów*. W zależności od szybkości tworzenia nowych *wątków rozproszonych* oraz częstotliwości odświerzania informacji o obciążeniu węzłów lokalne dane mogą szybko stać się nieaktualne. Można ten efekt złagodzić poprzez zwiększanie zapamiętanego obciążenia o wysłane na dany węzeł wątki.

4.5.2 Szeregowanie zadań za pomocą algorytmu cyklicznego

W pewnych zastosowaniach pewne zrównowarzenie obciążenia można osiągnąć poprzez zastosowanie algorytmu cyklicznego. Algorytm ten nie wymaga pobierania informacji o obciążeniu węzłów z *usługi odnajdywania węzłów*, przez co posiada mniejszy narzut komunikacyjny od algorytmów wymagających tych informacji.

4.6 Pamięć rozproszona

Ponieważ celem pracy nie było kompletne implementowanie pamięci rozproszonej, skupiono się na wyborze istniejącego rozwiązania, które spełniałoby wymagania zaprezentowane w punkcie 3.6.

4.6.1 Rozważane rozwiązania

Początkowo pod uwagę brane były następujące aplikacje:

- Memcached [?] – system dostarczający rozproszoną pamięć przechowującą dane w postaci par klucz-wartość. Nie znaleziono aktywnie rozwijanego klienta dla platformy .NET,
- Riak [?] – implementacja Amazon Dynamo [?], odrzucony ze względu na brak API dla .NET Framework,
- Polyphony [?, ?, ?, ?] – eksperymentalny projekt rozproszonej tablicy haszowej napisany w języku F#, nie wybrano ze względu na wczesnorozwojowy charakter projektu,

- Rhino DHT [?] – implementacja rozproszonej tablicy haszowej w języku C#; posiada zależności od Rhino PHT (*persistent hash table*) [?] oraz Rhino Queues [?],

a w późniejszym okresie również:

- Redis [?] – system rozpowszechniany na licencji *open source* przechowujący dane typu klucz-wartość.

Pierwszym wybranym rozwiązaniem było Rhino DHT, które było aktywnie rozwijane przez rozpoznawalnego autora – Ayende Rahiena. Niestety, okazało się, że wersjonowanie danych nie zostało w pełni zaimplementowane i nie było możliwe wykonanie atomowej operacji „odczytaj i zapisz", która była niezbędna do zrealizowania *zamków rozproszonych*. Szczegóły tego problemu zostały przedstawione w punkcie **??**.

Kolejnym rozwiązaniem wziętym pod uwagę był Redis – system autorstwa Salvatore Sanfilippo oraz Pietera Noordhuisa. Jest używany na co dzień jako mechanizm pamięci podręcznej wielu serwisów internetowych (m. in. cała rodzina StackExchange) – posiada przez to duże wsparcie i aktywnie działającą społeczność. Redis potrafi działać zarówno jako pojedynczy proces jak i w trybie master-slave, ponadto rozwijana jest wersja rozproszona – Redis Cluster. System ten posiada biblitekę dla .NET Framework dostarczoną przez firmę StackExchange. Redis został napisany dla rodziny systemów operacyjnych Linux, powstał jednak port tego systemu do systemu operacyjnego Windows. Problemy, które napotkano podczas uruchamiania usługi Redis (opisane w punkcie **??**) udało się rozwiązać, przez co system ten został wykorzystany w ostatecznej wersji pracy i w testach.

4.6.2 Interfejsy pamięci rozproszonej

Definicje *pamięci rozproszonej* oraz *rozszerzonej pamięci rozproszonej* zostały sformalizowane w formie interfejsów odpowiednio `IStorage` oraz `IExtendedStorage`. Interfejs `IStorage` wymaga implementacji metod do operacji na pojedynczych wartościach: `Store`, `StoreOrUpdate`, `Update`, `Retrieve` i `Remove` oraz operacji zbiorczych: `BulkStore`, `BulkStoreOrUpdate`, `BulkUpdate`, `BulkRetrieve` i `BulkRemove`. Semantyka tych operacji odpowiada operacjom zdefiniowanym w punkcie 3.6. Wszystkie operacje zdefiniowane w *pamięci rozproszonej* i *rozszerzonej pamięci rozproszonej* pozwalają na bezpieczne współbieżne wywołanie z wielu wątków (ang. *thread-safe*).

Interfejs `IExtendedStorage` rozszerza podstawowy interfejs `IStorage` o operacje pobrania i zwolnienia *zamków rozproszonych*: `AcquireLock` (w wersji z limitem czasu oczekiwania na pobranie zamka i bez) oraz `ReleaseLock`.

4.6.3 Rozproszone struktury danych i obiekty

Struktury danych i obiekty, które mogą być współdzielone między *wątkami rozproszonymi* zaimplementowane zostały w oparciu o *pamięć rozproszoną*, zdefiniowaną za pomocą interfejsu `IExtendedStorage` – pozwoliło to zastosować *zamki rozproszone* w celu zapewnienia poprawności przetwarzania – współbieżny dostęp do rozproszonych struktur danych i obiektów może być prowadzony zarówno z wątków jak i *wątków rozproszonych*. Każdy obiekt jest identyfikowany przez klucz będący łańcuchem znaków. W przypadku listy czy słownika na podstawie klucza wywiedzione zostają identyfikatory obiektów składających się na daną strukturę – zamków, metadanych i poszczególnych wartości.

Lista

Lista implementuje standardowy generyczny interfejs `IList<T>` z przestrzeni nazw `System.Collections.Generic`. Próba pobrania enumeratora zwraca obiekt klasy `DistributedListEnumerator`, który umożliwia iterowanie po kolekcji. *Lista rozproszona* została rozszerzona o operację `CopyPartTo` – jest to odpowiednik operacji `CopyTo` (efektywnej operacji kopiowania całej zawartości listy do wskazanej tablicy), który pozwala skopiować wybrany fragment listy w sposób efektywny i atomowy. Operacja ta jest szczególnie przydatna przy przetwarzaniu rozproszonym, gdzie dane znajdują się w wolnej *pamięci rozproszonej* (w stosunku do pamięci podręcznej), a każdy *wątek rozproszony* przetwarza fragment danych.

Słownik

Słownik implementuje standardowy generyczny interfejs `IDictionary<TKey, TValue>` z przestrzeni nazw `System.Collections.Generic`. Próba pobrania enumeratora zwraca obiekt klasy `DistributedDictionaryEnumerator`, który umożliwia iterowanie po kolekcji.

Licznik

W wielu scenariuszach (przykładowy scenariusz opisany w 3.6.1) przydatnym obiektem może być *licznik rozproszony*. Zaimplementowana klasa `DistributedCounter` udostępnia następujące operacje:

- `GetValue` – pobiera aktualną wartość licznika,
- `SetValue` – ustawia podaną liczbę jako wartość licznika,
- `Increase` – zwiększa wartość licznika o podaną wartość,
- `Decrease` – zmniejsza wartość licznika o podaną wartość,

- `GetAndIncrease` – atomowo pobiera aktualną wartość licznika i zwiększa ją o wskazaną liczbę, gdzie liczba o którą ma zostać zwiększony licznik może być ujemna.

4.6.4 Zamki rozproszone

Odpowiednikiem definicji *zamków rozproszonych* z punktu 3.6.3 jest interfejs `IStorageLock`. Zdefiniowane w nim zostały następujące operacje:

- `Acquire` – operacja pobrania zamka. Posiada wariant, który pozwala określić czas oczekiwania na pobranie zamka – jeśli czas ten zostanie przekroczony przed pobraniem zamka użytkownik dostanie informację o niepowodzeniu i będzie mógł kontynuować przetwarzanie. Operacja pobrania zamka bez podania czasu oczekiwania jest blokująca – przetwarzanie będzie kontynuowane dopiero po pobraniu zamka,
- `Release` – operacja zwolnienia zamka,
- `Wait` – wykonanie tej operacji powoduje rozpoczęcie oczekiwania na sygnał (ang. *pulse*) od innego procesu. Wykonanie tej operacji powoduje tymczasowe zwolnienie dostępu do zamka. Po otrzymaniu sygnału proces próbuje ponownie pobrać zamek. Podobnie jak `Acquire` operacja ta posiada wariant, który pozwala określić czas oczekiwania na sygnał,
- `Pulse` i `PulseAll` – wykonanie tych operacji powoduje wysłanie sygnału do procesów oczekujących na operacji `Wait`. W pierwszym przypadku sygnał dotrze do conajmniej jednego oczekującego procesu, a w drugim do wszystkich oczekujących procesów.

Interfejs `IStorageLock` dziedziczy z interfejsu `IDisposable` – sprawia to, że zamek można wykorzystać w podobny sposób jak słowo kluczowe `lock`, korzystając w tym celu z bloku `using`, co przedstawiono na listingu **??**. Cechą bloku `using` jest automatyczne wywołanie na jego końcu operacji `Dispose`, która w przypadku *zamków rozproszonych* powoduje zwolnienie zamka.

Listing 4.8: Realizacja sekcji krytycznej z wykorzystaniem zamka rozproszonego

```
1 using(var @lock = storage.AcquireLock("sampleLock"))
2 {
3   // sekcja krytyczna
4 }
```

4.7 Logowanie zdarzeń

Za zbieranie informacji na temat zdarzeń zachodzących w systemie odpowiada klasa `Log` udostępniająca m. in. statyczne metody:

- `ExceptionMessage` – do logowania wyjątków,
- `TraceMessage` – do logowania pozostałych zdarzeń.

Istnieje możliwość przekierowania wszystkich informacji do *pamięci rozproszonej*. W tym celu należy ustawić flagę `WriteToDistributedMemory` oraz uzupełnić nazwę hosta *pamięci rozproszonej* (`DistributedMemoryHost`), do którego ma odbywać się zapis. Warto zwrócić uwagę, że tryb pracy ze zbieraniem historii zdarzeń w *pamięci współdzielonej* może istotnie ograniczać wydajność systemu. W celu zmaterializowania zebranego logu udostępniona została metoda `SaveXes`, która zapisuje wszystkie zgromadzone w *pamięci rozproszonej* zdarzenia do pliku XML w formacie OpenXES.

Implementacja zapisu zdarzeń do pliku XML została wykonana na podstawie zmodyfikowanych plików XSD (pkt. 2.4.2) udostępnionych wraz z biblioteką OpenXES [?] w wersji 2.0 (przyczyny i sposób modyfikacji został opisany w punkcie ??). Szkielet klas został wygenerowany przy użyciu narzędzia XML Schema Definition Tool [?] (`xsd.exe`) dostarczanego wraz ze środowiskiem programistycznym .NET Framework 4.5.1. Przykład użycia programu `xsd.exe` został zaprezentowany na listingu ??.

Listing 4.9: Skrypt generujÂący klasy w jĂŞzyku C# na podstawie pliku XSD dla formatu OpenXES

```
1 "c:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools\xsd.
    exe" xes.xsd /classes /o:../Bluepath/Reporting/
2 "c:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools\xsd.
    exe" xesext.xsd /classes /o:../Bluepath/Reporting/
```

Wszystkie zdarzenia zachodzące w systemie zostały pogrupowane w tkw. *aktywności* (ang. *activity*). Rodzaje aktywności zachodzących w systemie zostały zdefiniowane w formie typu wyliczeniowego `Bluepath.Reporting.Log.Activity`, obejmuje on m. in.:

- `Service_is_ready` – zgłoszenie przez węzeł gotowości do przyjęcia zleceń,
- `Local_executor_started_running_user_code` – rozpoczęcie wykonywania kodu użytkownika w ramach rozproszonego wątku,
- `Local_executor_finished_running_user_code` – zakończenie wykonywania kodu użytkownika na węźle,
- `Sending_callback_with_result` – wysłanie wywołania zwrotnego z wynikiem działania wątku.

Wszystkie zdarzenia, które są mało istotne dla analizy procesu mogą być grupowane w ramach aktywności `Info` i powinny zostać odfiltrowane w pierwszej fazie analizy. Jeżeli użytkownik chce użyć własnej nazwy dla zdarzenia, może to zrobić korzystając z aktywności `Custom`, a właściwą nazwę przekazać jako parametr `message` do metody logowania zdarzeń.

Format XES definiuje dla zdarzeń w logu *zasób* (ang. *resource*) – w przypadku biblioteki *Bluepath* wartością tego pola jest zawsze wykonawca, który dokonał wpisu, korzystając z jego unikalnego identyfikatora `eid`. Implementacja zakłada możliwość skrócenia zapisu do n ostatnich znaków składających się na identyfikator w celu ułatwienia analizy przez człowieka. Może to potencjalnie wpłynąć na fałszywe sklasyfikowanie różnych zasobów jako tego samego w wyniku kolizji tak skonstruowanych identyfikatorów.

Zależności czasowe są istotną częścią analizy przebiegu procesu. Z wykorzystaniem serwerów czasu implementujących protokół NTP (ang. *network time protocol*) [**?**] możliwe jest zsynchronizowanie zegarów węzłów w klastrze z dokładnością do milisekund (przy dużych odległościach od serwera czasu – dziesiątek milisekund) [**?**]. Może to być niewystarczające do jednoznacznego określenia porządku zdarzeń. Klasa `Log` zawiera flagę `monotonicallyIncreasingLogTime`, której ustawienie uniemożliwia zapisanie dwóch zdarzeń z tym samym znacznikiem czasowym poprzez realizowanie dodatkowych operacji podczas zapisu do logu:

- pobranie *zamka rozproszonego*,
- odczyt ostatnio zapisanej wartości znacznika czasowego z *pamięci rozproszonej*,
- w przypadku gdy lokalna wartość znacznika jest mniejsza lub równa odczytanemu, jest ona zamieniana na odczytaną wartość zwiększoną o 1 milisekundę,
- zapis bieżącej wartości znacznika czasowego do *pamięci rozproszonej*,
- zwolnienie *zamka rozproszonego*.

4.8 Interfejs do komunikacji z systemem

Aby skorzystać z funkcji udostępnianych przez system (jak np. pobranie identyfikatora *wykonawcy* czy dostęp do *pamięci rozproszonej*) wewnątrz *wątku rozproszonego*, użytkownik musi uzyskać przeznaczony do tego obiekt. System automatycznie wstrzykuje go do metod jako jeden z parametrów – wystarczy, by był on typu `IBluepathCommunicationFramework`. Użytkownika ma również możliwość zapisywania własnych zdarzeń zachodzących w aplikacji do logu z użyciem opisanej w punkcie **??** statycznej klasy `Log`.

4.9 Dystrybucja aplikacji w klastrze

PowerShell Remoting [?] to usługa umożliwiająca wykonanie na zdalnych maszynach pojedynczych komend lub stworzenie pełnej zdalnej sesji PowerShell. Automatyzacja procesu dystrybucji plików binarnych systemu w klastrze została zrealizowana za pomocą zestawu skryptów: `Send-Folder` do przesyłania całych folderów i `Send-File` do przesyłania pojedynczych plików, z którego korzysta ten pierwszy.

Skrypt do wysyłania pojedynczych plików został zaczerpnięty z książki [?]. Przyjmuje 3 parametry: ścieżkę do pliku źródłowy znajdującego się na lokalnej maszynie, ścieżkę docelową na zdalnej maszynie oraz referencję do obiektu zdalnej sesji. Plik jest wczytywany do pamięci jako tablica bajtów, następnie jego transfer odbywa się strumieniowo w blokach o rozmiarze 1 MB. Po zrekonstruowaniu tablicy bajtów na zdalnej stronie jest ona zapisywana na dysk we wskazanej lokalizacji.

Skrypt do wysyłania folderów przedstawiony na listingu **??** przyjmuje jako parametry:

- adres zdalnego komputera (`-server`),
- opcjonalnie port, na którym nasłuchuje usługa PowerShell Remoting (`-port`),
- nazwę użytkownika (`-user`),
- hasło (`-password`),
- ścieżkę do folderu źródłowego na lokalnej maszynie (`-source`),
- oraz ścieżkę do folderu docelowego na zdalnej maszynie (`-destination`).

Skrypt tworzy obiekt zdalnej sesji uwierzytelniając się poprzez podaną nazwę użytkownika i hasło. W celu uproszczenia etapu konfiguracji środowiska, do wywołania metody `New-PSSession` można dodać przełącznik `-SessionOption` z parametrem `New-PSSessionOption -SkipCACheck`. Opcja ta powoduje jednak obniżenie poziomu bezpieczeństwa poprzez dopuszczenie niezaufanych certyfikatów maszyn. Następnie pobierana jest lista plików we wskazanym lokalnym folderze i dla każdego z plików wywoływany skrypt `Send-File`. Pomijane przy tym są pliki z symbolami (`.pdb`), ponieważ ich rozmiar jest znaczący w stosunku do rozmiaru plików samej aplikacji, a nie było konieczne podłączanie *debuggera* do procesów pracujących na zdalnych maszynach. Po zakończeniu przesyłania plików zdalna sesja jest zamykana. Warto zauważyć, że skrypt `Send-Folder` można wywołać wielokrotnie przekazując w parametrze -server kolejne adresy maszyn, aby rozdystrybuować pliki w całym klastrze.

Listing 4.10: Skrypt przesyÂłajÂący pliki ze wskazanego folderu na zdalnÂą maszynĂŞ

```
1 param (
2   [string]$server = $(throw "-server is required."),
3   [int]$port = 5986,
4   [string]$user = $(throw "-user is required."),
```

```
5   [string]$password = $(throw "-password is required."),
6   [string]$source = $(throw "-source is required."),
7   [string]$destination = $(throw "-destination is required.")
8 )
9
10 $secPassword = ConvertTo-SecureString $password -AsPlainText -Force
11 $credential = New-Object System.Management.Automation.PSCredential($user, $secPassword)
12 $uri = New-Object System.Uri("https://" + $server + ":" + $port)
13
14 $session = New-PSSession -ConnectionUri $uri -Credential $credential
15
16 Get-ChildItem -Path $source -File | Foreach-Object {
17   if ($_.Extension -ne ".pdb") {
18     $target = $destination + $_.Name
19     .\Send-File.ps1 $_.FullName $target $session
20   }
21 }
22
23 Disconnect-PSSession $session
```

Rozdział 5

Przykładowe zastosowania

W tym rozdziale przedstawione zostaną przykladowe zastosowania systemu Bluepath. Na bazie *wątków rozproszonych*, *pamięci rozproszonej* i nadbudowanych nad nią abstrakcji listy i słownika przygotowane zostały projekty:

- **DLINQ** (ang. *Distributed Language Integrated Query*) – dostawca metod manipulacji kolekcji w sposób zgodny z obecnym już w środowisku .NET dla standardowych, nierozproszonych kolekcji,
- **MapReduce** – abstrakcja specjalizowana do uruchamiania zadań w modelu mapowania i redukcji z możliwością zlecania zadań bez konieczności ponownej kompilacji systemu i dystrybuowania nowej wersji plików wykonywalnych na węzły,
- **System uzupełniania wyrazów** – system przygotowujący na podstawie podanej kolekcji dokumentów bazę prefiksów, która jest wykorzystywana do realizacji systemu podpowiedzi słów na podstawie pierwszych liter wpisywanych przez użytkownika,
- **Obliczanie przybliżenia liczby** π – algorytm obliczający przybliżoną wartość liczby π, który umożliwił przeprowadzenie testów porównawczych szybkości obliczeń w klastrze bez wykorzystania *pamięci rozproszonej* w stosunku do obliczeń prowadzonych na pojedynczym węźle.

5.1 DLINQ

Jednym z przykładowych zastosowań biblioteki Bluepath jest Distributed Language Integrated Query (DLINQ) – zbiór metod roszerzających możliwości operowania na kolekcjach bazowanych na *pamięci rozproszonej*. Obecna implementacja służy jako przykład (ang. *proof of concept*) wykorzystania możliwości dostarczanych przez bibliotekę Bluepath i dlatego dostarcza implementację podzbioru metod standardowego zestawu LINQ:

- `Select` – operacja ta pozwala przetransformować obiekt, np. wybierając jedynie podzbiór jego atrybutów, lub zmieniając typ obiektu,
- `SelectMany` – operacja podobna do `Select` z tą różnicą, że pojedynczy obiekt w wyniku transformacji zmieniany jest w kolekcję obiektów, a powstałe w ten sposób kolekcje są łączone w jedną,
- `Where` – operacja ta pozwala wyfiltrować te obiekty, które spełniają podany warunek,
- `GroupBy` – operacja pozwalająca grupować obiekty według zadanego klucza,
- `Count` – operacja zliczające obiekty spełniające zadany warunek.

Operacje te mogą być wykonywane zarówno na dostarczonej z biblioteką Bluepath liście rozproszonej (opisanej w 3.6.1) jak i na wszystkich obiektach implementujących interfejs `IEnumerable`. W drugim przypadku kolekcja źródłowa zostanie najpierw przetransformowana do listy rozproszonej, a następnie poddana dalszemu przetwarzaniu.

Sposób wykorzystania rozszerzeń DLINQ jest wzorowany na Parallel Language Integrated Query (PLINQ). Aby poddać kolekcję przetwarzaniu rozproszonemu wykonujemy najpierw metodę rozszerzającą `AsDistributed`. Metoda ta posiada jeden obowiązkowy argument – obiekt zapewniający dostęp do *pamięci rozproszonej*. Następnie na otrzymanym z tej metody obiekcie można wykonywać operacje zdefiniowane w DLINQ. Należy jednak mieć na uwadze, że operacje te podlegają założeniom biblioteki Bluepath i funkcje podawane jako argumenty muszą być funkcjami statycznymi. Na listingu **??** przedstawiono przykładowy kod poddający kolekcję transformacji przy użyciu metody `Select`.

Listing 5.1: Transformacja kolekcji przypomocy metod dostarczonych w DLINQ

```
1 var inputCollection = new List<double>(100);
2 ...
3 var processedCollection = (from num in inputCollection.AsDistributed(storage)
    select Math.Sqrt(num)).ToList();
```

W implementacji DLINQ można wyróżnić 3 podstawowe klasy: `DistributedQuery`, `DistributedEnumerable` oraz `UnaryQueryOperator`. Zostały one dokładniej opisane w poniższych punktach.

5.1.1 DistributedQuery

W celu zapewnienia jednoznacznego rozróżnienia pomiędzy metodami LINQ, oraz DLINQ wprowadzono klasę `DistributedQuery`. Klasa ta częściowo implementuje interfejs `IEnumerable` zapewniając powiązania pomiędzy metodami tego interfejsu. Stanowi ona bazę z której wywodzą się wszystkie klasy realizujące poszczególne wywołania DLINQ (np. `Select`) co pozwala na wykonywanie tych samych metod na instancjach różnych klas składowych DLINQ.

5.1.2 DistributedEnumerable

Statyczna klasa `DistributedEnumerable` zawiera wszystkie metody rozszerzające DLINQ. Każda z tych metod weryfikuje parametry wejściowe w celu przekazania użytkownikowi zrozumiałych informacji w przypadku podania błędnych parametrów. Następnie tworzona jest klasa realizująca wywołanie DLINQ i zwracana użytkownikowi w celu dalszego przetwarzania (samo wykonanie podobnie jak w LINQ następuje dopiero wtedy gdy potrzebne są wyniki np. wywołanie metody `ToList`, lub enumeracja).

Specjalną metodą rozszerzającą zaimplementowaną w klasie `DistributedEnumerable` jest metoda `AsDistributed`. Służy ona przygotowaniu kolekcji wejściowej do przetwarzania przez metody DLINQ. Jest to realizowane poprzez opakowanie kolekcji wejściowej klasą `DistributedEnumerableWrapper`, która przenosi zawartość kolekcji wejściowej do *listy rozproszonej* (wyjątkiem jest sytuacja, w której kolekcja wejściowa jest *listą rozproszoną*).

5.1.3 UnaryQueryOperator

Klasą bazową dla klas implementujących metody DLINQ jest `UnaryQueryOperator`. Jest to klasa, która gromadzi wszystkie metody wspólne dla metod DLINQ takich jak tworzenie *wątków rozproszonych* z wykorzystaniem ustawień przekazanych przez użytkownika w metodzie `AsDistributed`, czy zebranie wyników przetwarzania z poszczególnych *wątków rozproszonych*. Nazwa `UnaryQueryOperator` wynika z faktu, że wszystkie metody DLINQ oparte na tej klasie operują na jednej kolekcji. W przypadku implementacji metod takich jak `Union` – metoda odpowiadająca za połączenie dwóch kolekcji – należałoby stworzyć kolejne klasy, np. `BinaryQueryOperator`.

5.2 MapReduce

W celu porównania pracochłonności implementacji aplikacji z użyciem bibliteki Bluepath i bez niej przygotowane zostały aplikacje do realizacji przetwarzania w modelu mapowania i redukcji zarówno w środowisku Bluepath jak i bezpośrednio z wykorzystaniem platformy Windows Communication Foundation [**?**]. Założeniem aplikacji było, by kod metod `Map` i `Reduce` dostarczany był jako kod źródłowy w języku C# w formie klasy implementującej odpowiedni interfejs – `IMapProvider` lub `IReduceProvider` – przedstawiony na listingu **??** i składowany razem z danymi w *pamięci rozproszonej*. Kompilacja kodu miała odbywać się na węzłach bezpośrednio przed wykonaniem metody, do zrealizowania tego celu wybrana została biblioteka CS-Script [**?**].

Listing 5.2: Interfejsy IMapProvider i IReduceProvider

```
1 public interface IMapProvider
2 {
3    IEnumerable<KeyValuePair<string, string>> Map(string key, string value);
4 }
5
6 public interface IReduceProvider
7 {
8    KeyValuePair<string, string> Reduce(string key, IEnumerable<string> values);
9 }
```

5.2.1 Koordynator

Główną klasą w aplikacji jest *koordynator*, który pełni rolę nadzorcy zadań (z tego powodu nazywany jest także *JobTrackerem*). Jego zadaniem jest przydzielanie zdań mapowania i redukcji podległym węzłom. Poniżej opisano jego implementacje z wykorzystaniem biblioteki Bluepath jak i w oparciu bezpośrednio o technologię WCF.

Koordynator oparty o Bluepath

W środowisku Bluepath jego zadanie sprowadzało się w pierwszej fazie przetwarzania do uruchomienia *wątków rozproszonych* w liczbie odpowiadającej liczbie plików wejściowych, które były inicjowane metodą, która:

- ładowała kod metody `Map` z *pamięci rozproszonej*,
- wykonywała skompilowany kod *mappera* na wskazanym fragmencie danych wejściowych z *pamięci rozproszonej*,
- zapisywała wygenerowane pary klucz-wartość do *pamięci rozproszonej*,
- zwracała listę kluczy jako wynik wykonania *wątku rozproszonego* do *koordynatora*.

Zainicjowane w ten sposób wątki były następnie uruchamiane z parą parametrów – nazwą pliku zawierającego dane do przetworzenia oraz nazwą pliku z kodem źródłowym metody `Map`. W drugiej fazie przetwarzania tworzonych było tyle rozproszonych wątków, ile kluczy było wynikiem pierwszej fazy. Każdy z nich inicjowany był metodą, która:

- ładowała kod metody `Reduce` z *pamięci rozproszonej*,
- wykonywała skompilowany kod *reducera* na wskazanym fragmencie danych wejściowych z *pamięci rozproszonej*,
- zapisywała wygenerowane pary klucz-wartość do *pamięci rozproszonej*,

- zwracała listę kluczy jako wynik wykonania rozproszonego wątku do *koordynatora*.

Przygotowywana jest lista przydziału kluczy do przetwarzania poszczególnym wątkom dokonującym redukcji, po czym są one uruchamiane z parą parametrów – kluczem oraz nazwą pliku z kodem źródłowym metody `Reduce`.

Koordynator w technologii WCF

W projekcie bez użycia biblioteki Bluepath koordynator jest opisany za pomocą interfejsu definiującego kontrakt usługi (`ServiceContract`) i udostępnia końcówkę WCF (ang. *WCF endpoint*). Interfejs ten został przedstawiony na listingu **??**. W tej implementacji koordynator, oprócz zlecania wykonania zadań, musi również zająć się innymi zadaniami związanymi z obsługą klastra:

- rejestrowywaniem i wyrejestrowywaniem węzłów, udostępnianiem ich listy (metody `AddWorker`, `RemoveWorker`, `GetWorkers`),
- udostępnianiem wyników przetwarzania, zarówno pełnych jak i częściowych, jeżeli przetwarzanie jeszcze trwa (`GetResults`),
- udostępnianiem interfejsu do manipulacji danymi znajdującymi się w jego części pamięci – zapisywanie, usuwanie, pobieranie listy plików, czyszczenie pamięci (`AddToStorage`, `RemoveFromStorage`, `ListStorageFiles`, `CleanStorage`).

Pliki, które mają być przetwarzane oraz kod metod Map i Reduce muszą w pierwszej kolejności trafić do pamięci koordynatora. Poszczególne pliki są następnie wysyłane do konkretnych węzłów, którym przydzielone zostało zadanie ich przetwarzania. Podobnie po zakończeniu fazy mapowania koordynator przygotowuje plan wykonania fazy redukcji i rozsyła w klastrze listę przydziału kluczy do węzłów. Na tej podstawie węzły mogą przesłać między sobą wyniki pośrednie. Po zakończeniu przetwarzania węzły przesyłają pliki wynikowe do pamięci koordynatora.

Listing 5.3: Interfejs ICoordinatorService

```
1  [ServiceContract]
2  public interface ICoordinatorService
3  {
4      [OperationContract]
5      void AddWorker(Uri uri);
6
7      [OperationContract]
8      Uri[] GetWorkers();
9
10     [OperationContract]
11     void RemoveWorker(Uri uri);
12
```

```
13      [OperationContract]
14      bool RunJob(int numberOfMappers, int numberOfReducers, Uri mapCodeFile, Uri
            reduceCodeFile, Uri[] filesToProcess);
15
16      [OperationContract]
17      Uri AddToStorage(string fileName, string content);
18
19      [OperationContract]
20      Uri[] ListStorageFiles();
21
22      [OperationContract]
23      void RemoveFromStorage(Uri uri);
24
25      [OperationContract]
26      void CleanStorage();
27
28      [OperationContract]
29      MapReduceResult GetResults();
30  }
31
32  [DataContract]
33  public class MapReduceResult
34  {
35      [DataMember]
36      public Tuple<string, string>[] KeysAndValues { get; set; }
37
38      [DataMember]
39      public bool IsRunning { get; set; }
40  }
```

5.2.2 Pamięć masowa

Aplikacja ta korzysta z abstrakcji pamięci masowej, którą definiuje interfejs `IMapReduceStorage`. Został on zaprezentowany na listingu **??**. Udostępnia on dodatkowe operacje, które umożliwiają np. pobieranie listy istniejących plików czy listy kluczy wygenerowanych w wyniku fazy mapowania. Interfejs ten jest wspólny dla implementacji z użyciem środowiska Bluepath – wynikiem jest klasa `BluepathStorage` wykorzystująca *pamięć rozproszoną* – jak i bez niego – w tym przypadku jest to `FileSystemStorage` bazujący na systemie plików. Klasa `FileSystemStorage` przed zapisaniem pliku na dysk zamienia jego nazwę na reprezentację w kodowaniu **Base64**. Mogło więc się zdarzyć, że w naziwe pliku pojawi się symbol „+”. Po zapisaniu tak zakodowanego klucza za pomocą klasy `System.Uri` następowała zamiana znaku „+” na spację, przez co wartość przestawała być poprawnym ciągiem zgodnym z ko-

dowaniem Base64. Jako rozwiązanie tego problemu zastosowano zamianę spacji z powrotem na znaki „+” w momencie odczytu klucza z otrzymanego `Uri`.

Listing 5.4: Interfejs IMapReduceStorage

```
1  public interface IMapReduceStorage
2  {
3      IEnumerable<Uri> ListFiles();
4      string Read(string fileName);
5      string Read(Uri uri);
6      string[] ReadLines(string fileName);
7      void Store(string fileName, string value);
8      void Store(Uri uri, string value);
9      void Clean();
10     string GetFileName(Uri uri);
11     IEnumerable<string> GetKeys();
12     void Remove(Uri uri);
13 }
```

5.2.3 Węzeł obliczeniowy w technologii WCF

Usługa węzła obliczeniowego jest komponentem obecnym jedynie w wersji aplikacji zbudowanej w oparciu o technologię WCF (środowisko Bluepath zapewnia możliwość uruchamiania procesów roboczych na węzłach bez dodatkowego nakładu pracy ze strony programisty). Usługa ta zarządza procesami roboczymi na danym węźle oraz pełni rolę pośrednika między koordynatorem a procesami roboczymi. Węzły mogą komunikować się między sobą w celu bezpośredniego przesłania plików pomiędzy fazami mapowania i redukcji. Interfejs usługi umożliwia:

- utworzenie nowego procesu roboczego (metoda `Init`),
- uruchomienie przetwarzania w ramach jednego z procesów roboczych (`Map`, `Reduce`),
- sprawdzenie, czy przetwarzanie w danym procesie roboczym się zakończyło (`TryJoin`),
- zlecenie przesłania plików między węzłami roboczymi na podstawie listy przydziału kluczy do węzłów (`TransferFiles`),
- zapisanie pliku w pamięci procesu roboczego (`PushFile`),
- pobranie informacji o obciążeniu węzła (`GetPerformanceStatistics`) – zajętości pamięci operacyjnej, obciążeniu procesora, ilości wolnego miejsca na dysku.

Listing 5.5: Interfejs IRemoteWorkerService

```
1  [ServiceContract]
2  public interface IRemoteWorkerService
```

```
{
    [OperationContract]
    void Init(int workerId);

    [OperationContract]
    void Map(Uri uri, Uri mapFuncUri);

    [OperationContract]
    void Reduce(Uri uri, Uri reduceFuncUri);

    [OperationContract]
    string[] TryJoin(int workerId, Uri callbackUri);

    [OperationContract]
    Uri[] TransferFiles(int workerId, Dictionary<string, Uri> keysAndUris);

    [OperationContract]
    Uri PushFile(int workerId, string fileName, string content);

    [OperationContract]
    PerformanceMonitor.PerformanceStatistics GetPerformanceStatistics();

    Uri EndpointUri { get; }
}
```

5.2.4 Wyniki eksperymentu

Implementacja środowiska do przetwarzania w modelu mapowania i redukcji zajęła 6 dni dwóm programistom, natomiast z wykorzystaniem biblioteki Bluepath czas ten skrócił się do 2 dni i to przy zaangażowaniu tylko jednego programisty, a liczba linii kodu została zredukowana z 585 do 194.

5.3 System uzupełniania wyrazów

Wiele aplikacji operujących na dużej ilości danych próbuje wspomóc swoich użytkowników pozwalając filtrować dane przy pomocy zapytań tekstowych. Ponieważ formułowanie zapytań podlega często ograniczeniom, np. konieczności użycia konkretnych słów kluczowych, można stosować techniki wspomagające użytkownika w tym działaniu.

Jedną z takich technik jest system uzupełniania wyrazów (ang. *autocomplete*). W trakcie wpisywania zapytania użytkownikowi prezentowana jest lista zawierająca sugerowane zakończenia wpisywanego wyrazu. W przypadku gdy wyraz, który

użytkownik miał zamiar wpisać, znajduje się na liście może on zostać wybrany. W przeciwnym wpadku użytkownik musi kontynuować wpisywanie wyrazu.

System tego typu można zrealizować poprzez porównanie wprowadzonego słowa ze słowami występującymi w przetwarzanych danych. Ponieważ przetwarzanie wszystkich posiadanych danych w czasie rzeczywistym byłoby bardzo powolne, można je poddać wstępnej obróbce – np. wygenerowaniu i powiązaniu prefiksów z ich możliwymi rozwinięciami.

System uzupełniania wyrazów został zaimplementowany jako przykład zastosowania biblioteki *Bluepath*. Przetwarzanie zostało podzielone na trzy odrębne części:

- wczytywanie danych – wytworzenie i powiązanie prefiksów i ich rozwinięć – opisane w **??**,
- uzupełnianie wyrazów – demonstracja realizacji funkcji podpowiadania możliwych zakończeń prefiksu – opisane w **??**,
- czyszczenie stanu systemu – przywrócenie całego systemu do stanu początkowego bez konieczności ponownego uruchamiania aplikacji.

Na podział ten miały wpływ dwa czynniki:

- z założenia przygotowywanie prefiksów (wraz z rozwinięciami) miało być czynnością niezależną od wyszukiwania możliwych rozwinięć wpisanego wyrazu – system udziela odpowiedzi na podstawie najświeższych posiadanych danych,
- w trakcie pisania systemu okazało się, że przydatna jest metoda przywracająca system do stanu początkowego, a niebędąca częścią głównego przetwarzania.

5.3.1 Wczytywanie danych

Zaimplementowany system uzupełniania wyrazów, przed rozpoczęciem wyszukiwania możliwych zakończeń zadanego prefiksu, musi zostać zainicjalizowany. Dokumenty zawierające przykładowe słowa są wczytywane przez węzeł inicjalizujący operację wczytywania danych i dodawane jako elementy *listy rozproszonej*. Następnie każdy *wątek rozproszony* pobiera nieprzetworzony fragment listy i na podstawie zawartych w nim dokumentów uzupełnia słownik prefiksów i możliwych dla nich uzupełnień. Gdy na liście nie ma już elementów do przetworzenia, *wątek rozproszony* kończy przetwarzanie. Gdy wszystkie *wątki rozproszone* zakończą przetwarzanie, każdy z węzłów zawiera częściową informację dotyczącą prefiksów i ich uzupełnień. Fragment funkcji `LoadDocuments` odpowiedzialnej za operację wczytywania danych wykonywaną przez *wątek rozproszony* został przedstawiony na listingu **??**.

Listing 5.6: Fragment funkcji LoadDocuments

```
1 private static int LoadDocuments(string inputKey, string counterKey, int
      chunkSize, IBluepathCommunicationFramework bluepath)
2 {
```

```
3   var inputList = new DistributedList<string>(bluepath.Storage as
        IExtendedStorage, inputKey);
4   var inputCount = inputList.Count;
5   var counter = new DistributedCounter(bluepath.Storage as IExtendedStorage,
        counterKey);
6   int indexEnd = 0;
7   do
8   {
9     int noOfElements = chunkSize;
10    int indexStart = counter.GetAndIncrease(chunkSize);
11    [...]
12    var inputDocuments = new string[noOfElements];
13    inputList.CopyPartTo(indexStart, noOfElements, inputDocuments);
14    foreach (var document in inputDocuments)
15    {
16      var words = document.Split(' ');
17      foreach (var word in words)
18      {
19        [...] // wygenerowanie prefiksÃŁw i zapisanie ich w pamiÃŞci lokalnej
20      }
21    }
22  } while (indexEnd <= inputCount);
23
24  return 0;
25 }
```

5.3.2 Uzupełnianie wyrazów

Głównym celem systemu uzupełniania wyrazów jest wyszukiwania możliwych zakończeń zadanego prefiksu. W tym celu należy przeszukać informacje posiadane przez każdy z węzłów, a następnie zagregować uzyskane wyniki częściowe. Aby mieć pewność, że każdy węzeł zostanie przeszukany zastosowano planistę szeregującego zadania za pomocą algorytmu karuzelowego (opisanego w punkcie **??**) i uruchomiono tyle *wątków rozproszonych*, ile znajduje się węzłów w systemie. Inicjalizacja planisty została przedstawiona na listingu **??**.

Listing 5.7: Inicjalizacja planisty szeregujÂącego zadania za pomocÂą algorytmu karuzelowego

```
1 var services = connectionManager.RemoteServices.Select(s => s.Key).ToArray();
2 var scheduler = new RoundRobinLocalScheduler(services);
```

Po zakończeniu przetwarzania przez wszystkie wątki należy zagregować dane. Częścią tej operacji, oprócz połączenia wyników, jest usunięcie duplikatów, które mogą się pojawić ze względu na to, że w tej implementacji każdy *wątek rozproszony*

wykonuje przetwarzanie lokalnie (niezależnie od pozostałych wątków). Na listingu **??** przedstawiono zbieranie wyników oraz usunięcie duplikatów.

Listing 5.8: Zebranie wynikĂłw w systemie uzupeÂłniania wyrazĂłw

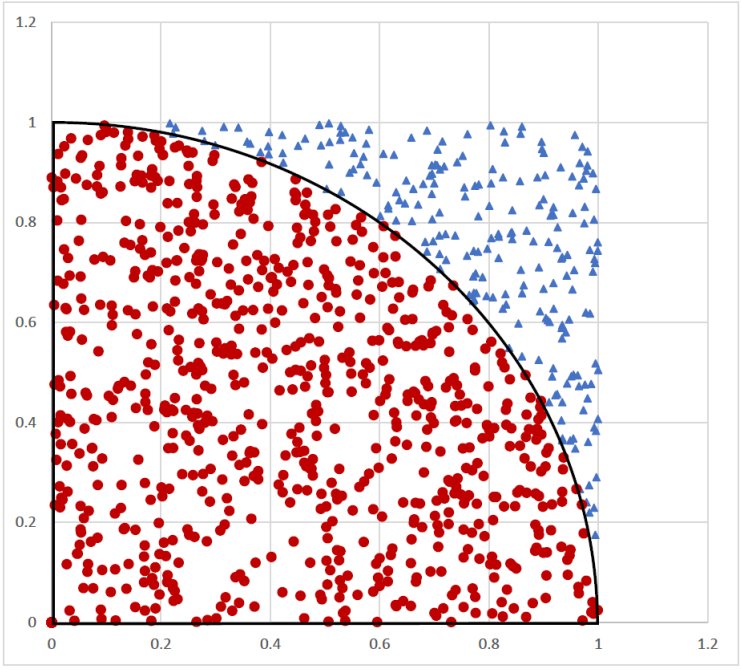
```
var joinedResult = new List<string>();
foreach (var thread in threads)
{
  thread.Join();
  joinedResult.AddRange(thread.Result as string[]);
}

var endResult = joinedResult.Distinct().ToArray();
```

5.4 Obliczanie przybliżenia liczby π

Metoda Monte Carlo jest formą symulacji z wykorzystaniem generatora liczb pseudolosowych. Każde kolejne uruchomienie algorytmu daje nieco inne wyniki, o ile nie zapewnimy generatorowi takich samych warunków początkowych (ang. *seed*). Obliczenie przybliżenia liczby π sprowadza się do losowania n razy pary liczb (x, y) z przedziału $[0, 1]$ i inkrementowaniu licznika m, gdy para spełnia warunek $x^2 + y^2 \leq 1$. W przykładowej symulacji dla $n = 1000000$ otrzymano $m = 785930$. Oznacza to, że m z n sprawdzonych punków znalazło się w obszarze reprezentującym $\frac{1}{4}$ powierzchni koła, którego środek znajduje się w punkcie $(0, 0)$ – patrz rysunek **??**. Ostateczne wyliczenie otrzymanego przybliżenia liczby π przedstawia równanie **??**.

$$p = \frac{m}{n} = \frac{785930}{1000000}, \quad p \times 4 = 3,14372 \approx \pi \tag{5.1}$$

Zwiększanie liczby próbek n nie musi poprawiać dokładności otrzymanego wyniku. W zależności od jakości zastosowanego generatora liczb pseudolosowych, może on w pewnym momencie zakończyć bieżącą sekwencję i rozpocząć zwracanie liczb z tej samej sekwencji od początku. Ponieważ celem tego testu nie jest zbadanie dokładności przybliżenia liczby π a porównanie szybkości obliczeń w klastrze i na



Rysunek 5.1: Ilustracja obliczania przybliżenia liczby π metodą Monte Carlo

pojedynczym węźle, jako generator została wybrana klasa `System.Random`[1].

Otrzymane wyniki i ich analiza zostały przedstawione w rozdziale *Testy wydajnościowe i jakościowe* w punkcie **??**.

[1]Implementacja klasy `System.Random` w .NET Framework 4.0 posiada kilka cech, które sprawiają, że nie nadaje się ona do zastosowań, w których oczekujemy ciągu o wysokim stopniu losowości.

- Inicjowanie generatora przez domyślny konstruktor używa aktualnego czasu. Z tego powodu utworzenie wielu jego instancji w krótkim czasie, np. w środowisku wielowątkowym, może prowadzić do otrzymania takich samych sekwencji liczb. Co więcej, przestrzeń inicjująca generator jest stosunkowo mała – mając już 2^{16} instancji generatora zainicjowanego losowymi wartościami istnieje duże prawdopodobieństwo otrzymania tej samej sekwencji wielokrotnie.
- Dla niektórych wartości parametrów metoda `Next` zwraca sekwencje, których rozkład nie jest równomierny.

W zastosowaniach, w których wymagane jest bezpieczeństwo zaleca się stosowanie implementacji opartych o metody kryptograficzne, np. `System.Security.Cryptography.RNGCryptoServiceProvider`.

Rozdział 6

Testy jakościowe i wydajnościowe

W bibliotekach wykorzystywanych przez programistów równie ważna jak bogata funkcjonalność czy łatwość użycia jest stabilność działania i pewność, że błąd w wykonaniu aplikacji nie leży po stronie biblioteki. W związku z tym istotne jest pokrycie kodu biblioteki zestawem testów zarówno jednostkowych jak i akceptacyjnych, które zostały opisane w niniejszym rozdziale.

Oprócz testów jakościowych przeprowadzone zostały testy wydajnościowe, które pozwoliły określić wydajność biblioteki w obecnym stanie implementacyjnym oraz określić wpływ rodzaju przetwarzania na zasadność prowadzenia przetwarzania rozproszonego.

6.1 Testy jakościowe

Testy jakościowe pozwalają na sprawdzenie konkretnych funkcji biblioteki w spreparowanym środowisku testowym. Testy jednostkowe i integracyjne, działają w środowisku Microsoft Unit Test Framework [?] zintegrowanym ze środowiskiem Visual Studio. Dodatkowo wykorzystana została biblioteka Shoudly [?] do zapisu asercji oraz Moq [?] do tworzenia atrap obiektów.

6.1.1 Testy jednostkowe

Testując niektóre funkcje biblioteki wskazane było sprawdzenie ich zachowania abstrahując od ich zależności. W tym celu równolegle z tworzonym kodem powstawały testy jednostkowe. Należy również zwrócić uwagę, że testy jednostkowe wykonują się znacznie szybciej od testów integracyjnych, co pozwala znacząco skrócić czas wykrycia oraz poprawy ewentualnych błędów. Poniżej przedstawiono przykładowe przypadki testowe:

- *lokalny wykonawca* – operacja `Join` z określonym limitem czasu oczekiwania na jej zakończenie,

- *zdalny wykonawca* – testy operacji `Pulse` z użyciem atrapy obiektu (ang. *mock object*),
- usługa *zdalnego wykonawcy* – wykonanie metody z użyciem zserializowanego uchwytu, przechwytywanie wyjątków w kodzie użytkownika, zwracanie statystyk obciążenia węzła oraz testy z użyciem zmienionej implementacji interfejsu (ang. *fake object*) – operacji `Join` (synchronicznej i asynchronicznej), rzucenia przechwyconego w kodzie użytkownika wyjątku w momencie wykonywania operacji `Join`, wstrzykiwania zależności `IBluepathCommunicationFramework` (przez podmianę oraz przez dodanie brakującego parametru),
- *wątek rozproszony* – wykonanie z użyciem *lokalnego wykonawcy*, *zdalnego wykonawcy* (potrzebne usługi, tj. `IRemoteExecutorService` i `IScheduler` zostały zasymulowane za pomocą atrap obiektów, użyto też *zarządcy połączeń* w trybie bez wywołań zwrotnych), głębokie kopiowanie parametrów wywołania,
- planista `ThreadNumberScheduler` – test sprawdza, czy następuje wybór najmniej obciążonego węzła spośród zasymulowanych,
- z uwagi na użycie własnej klasy `ServiceUri` jako klucza w słowniku, testowano też taki scenariusz.

6.1.2 Testy integracyjne

W celu przetestowania działania funkcji takich jak przesyłanie klas jako parametrów wskazane jest stworzenie testów integracyjnych. Ich zadaniem jest przetestowanie nie tylko logiki wykonywanego kodu, ale również wykrycie problemów jakie mogą się pojawić w związku z działaniem w środowisku sieciowym (np. wymagane uprawnienia do utworzenia usługi nasłuchującej, serializacja obiektów).

Jednym z problemów związanych z testami integracyjnymi jest potrzeba stworzenia środowiska testowego. W momencie gdy zachodzi potrzeba uruchomienia zewnętrznego programu (np. Redisa) system operacyjny nie dostarcza narzędzi, które pozwoliłyby określić nie tylko czy dany proces jest już uruchomiony, ale również czy skończył on proces inicjalizacji. Należy również zwrócić uwagę, że testy nie powinny być od siebie zależne – a więc efekt wykonania jednego testu nie powinien mieć wpływu na inne testy. Aby tego uniknąć, każdy test został wyposażony w taki zestaw parametrów, który pozwoli mu się wykonać niezależnie od innych (np. port na którym ma nasłuchiwać usługa *Bluepath*).

W ramach testów integracyjnych przetestowane zostały następujące aspekty:

- *wątki rozproszone* – zachowanie systemu w wyniku próby wykonania metod akceptujących różne typy parametrów i zwracające wyniki różnego typu – tablice, klasy (także generyczne), delegaty, a także obsługa wyjątków w kodzie użytkownika oraz w przypadku nieprawidłowego wywołania; test obejmował także wykonanie metody z biblioteki napisanej w języku F#,

- *zarządca połączeń* – pobieranie listy dostępnych węzłów z *usługi odnajdywania węzłów* i aktualizowanie swojej lokalnej kopii listy (dodawanie nowych węzłów i usuwanie wyrejestrowanych), pobieranie informacji o obciążeniu węzłów zarejestrowanych w klastrze,
- *pamięć rozproszona* – testy zapisu i odczytu obiektów (także jako operacji zbiorczych),
- *zamki rozproszone* – pobieranie i zwalnianie zamków (także z limitem czasu), budzenie wątków czekających na zamkach,
- a także testy struktur danych i obiektów zbudowanych na bazie *pamięci rozproszonej* – lista, słownik, licznik.

6.2 Testy wydajnościowe

Ważną częścią tworzenia biblioteki programistycznej jest zweryfikowanie jej zachowania w przykładowych zastosowaniach. Pozwala to zweryfikować słuszność przyjętych założeń i rozwiązań.

Przy okazji weryfikacji działania biblioteki *Bluepath* przeprowadzone zostały testy wydajnościowe, zestawiające efekt wykonania przetwarzania w klastrze w stosunku do przetwarzania na pojedynczym procesorze oraz porównanie dostarczonych przykładowych algorytmów szeregowania zadań.

6.2.1 Środowisko

Środowisko do przeprowadzenia testów wydajnościowych zostało przygotowane w oparciu o platformę Microsoft Azure [**?**]. Klaster składał się z 7 maszyn wirtualnych – na sześciu z nich działały usługi systemu Bluepath pod kontrolą systemu operacyjnego Windows Server 2012 R2 Datacenter (wydanego 17 czerwca 2014 r.), a na jednej maszynie uruchomiony został host *pamięci rozproszonej* Redis pod kontrolą systemu operacyjnego Ubuntu Server 14.04 LTS (wydanego 20 czerwca 2014 r.). Maszyny różniły się konfiguracją sprzętową w zależności od przeznaczenia:

- węzeł inicjujący przetwarzanie był typu *Medium VM (Basic A2)* i posiadał 2 wirtualne rdzenie AMD Operton 1,6 GHz i 3,5 GB pamięci operacyjnej,
- węzły obliczeniowe były typu *Small VM (Basic A1)* dysponowały po 1 wirtualnym rdzeniu AMD Opteron 1,6 GHz i 1,75 GB pamięci operacyjnej,
- węzeł hostujący usługę Redis była to instancja typu *Large VM (Basic A3)* posiadająca cztery rdzenie AMD Opteron 1,6 GHZ i 7 GB pamięci operacyjnej.

Szczegóły konfiguracji zestawiono w tabeli **??**. Maszyny wirtualne wchodzące w skład klastra połączone zostały w wirtualną sieć, w której wyłączone były zapory sieciowe,

a komunikacja odbywała się z użyciem prywatnych wewnętrznych adresów IP (tzw. *DIP*, przydzielony automatycznie przez Windows Azure z użyciem DHCP).

Do dystrybucji nowych wersji bibliotek systemu Bluepath na wszystkie maszyny w klastrze przygotowany został skrypt PowerShell przedstawiony na listingu **??**. Wykorzystuje on opisany w punkcie **??** skrypt `Send-Folder`.

Listing 6.1: Skrypt dystrybuujÂący biblioteki w klastrze

```
.\Send-Folder.ps1 -server [adres wĂŞzÂła 1] -user [nazwa uÂżytkownika]
  -password [hasÂło] -source [folder ÂźrĂłdÂłowy] -destination [folder docelowy
      ]
...
.\Send-Folder.ps1 -server [adres wĂŞzÂła n] -user [nazwa uÂżytkownika]
  -password [hasÂło] -source [folder ÂźrĂłdÂłowy] -destination [folder docelowy
      ]
```

6.2.2 Generowanie słownika prefiksów

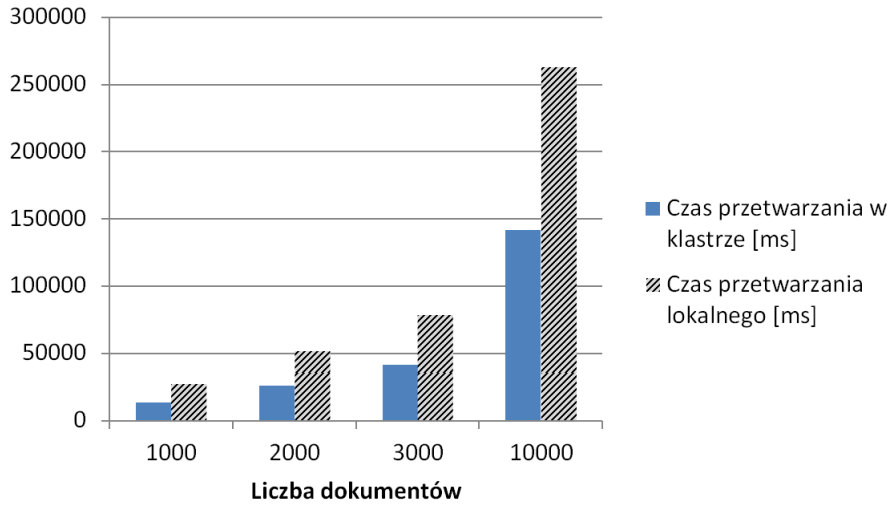
Jednym z algorytmów wykorzystanych do pomiaru wydajności był algorytm tworzący prefiksy na potrzeby przeprowadzenia operacji uzupełniania tekstu opisany w punkcie **??**. Dla określonej liczby dokumentów znajdujących się w *pamięci rozproszonej* pomiar czasu przetwarzania był powtarzany dziesięć razy. Wykres **??** przedstawia porównanie uśrednionych czasów generowania słownika prefiksów na pojedynczym węźle oraz w klastrze. Dla zbioru 10000 dokumentów udało się uzyskać prawie dwukrotne przyspieszenie w stosunku do przetwarzania na pojedynczej maszynie. Warto również nadmienić, że w algorytmie testującym wprowadzono mechanizm okresowo czyszczący lokalną pamięć prefiksów ze względu na wyczerpywanie się pamięci w przypadku przetwarzania na pojedynczym węźle.

6.2.3 Obliczanie przybliżenia liczby π

Kolejnym algorytmem, przy pomocy którego zmierzono zysk związany z zastosowaniem biblioteki Bluepath było obliczanie przybliżenia liczby π metodą przedstawioną

Tabela 6.1: Konfiguracja testowego klastra

Nazwa maszyny	Typ instancji	Liczba rdzeni	Pamięć RAM	System operacyjny
Master	A2	2	3,5 GB	Windows Server
Slave1	A1	1	1,75 GB	Windows Server
Slave2	A1	1	1,75 GB	Windows Server
Slave3	A1	1	1,75 GB	Windows Server
Slave4	A1	1	1,75 GB	Windows Server
Slave5	A1	1	1,75 GB	Windows Server
Redis	A3	4	7 GB	Ubuntu Server

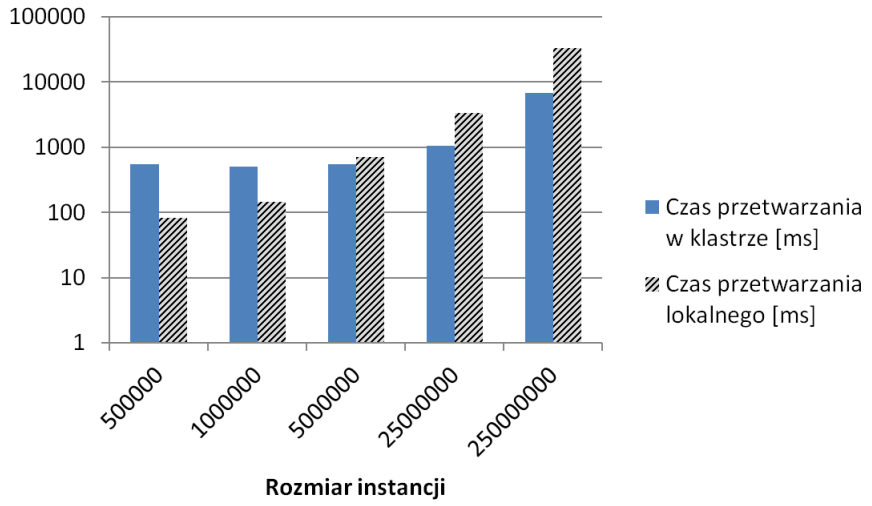


Rysunek 6.1: Porównanie czasu generowania słownika prefiksów

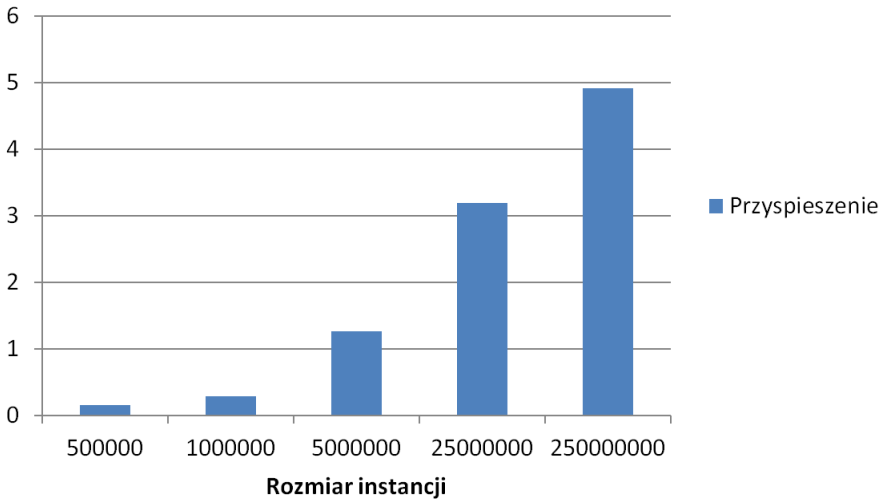
w punkcie **??**. Wykres **??** przedstawia porównanie uśrednionych czasów przetwarzania na pojedynczym węźle oraz w klastrze w zależności od rozmiaru instancji. Czas został przedstawiony w skali logarytmicznej.

Dla dużych instancji przetwarzanie przy pomocy biblioteki Bluepath jest dużo szybsze niż przetwarzanie na pojedynczym węźle. Jednak dla małych instancji narzut generowany przez bibliotekę sprawia, że przetwarzanie w klastrze staje się mniej efektywne niż przetwarzanie lokalne. Można to również zauważyć na wykresie **??**. Przedstawia on przyspieszenie uzyskane w związku z zastosowaniem biblioteki Bluepath do przetwarzania w klastrze w stosunku do przetwarzania na pojedynczym węźle.

Narzut widoczny na obu wykresach jest częstym zjawiskiem w systemach rozproszonych i wynika z konieczności przeprowadzenia komunikacji sieciowej.



Rysunek 6.2: Porównanie czasu obliczania przybliżenia liczby π



Rysunek 6.3: Przyspieszenie w stosunku do przetwarzania lokalnego

Rozdział 7

Podsumowanie

Niniejszy rozdział stanowi podsumowanie pracy. Omówiono w nim przebieg realizacji, napotkane w trakcie implementacji problemy oraz przedstawiono potencjalne kierunki dalszego rozwoju projektu.

7.1 Przebieg realizacji

W trakcie realizacji tej pracy udało się zrealizować główny cel – zaprojektowanie biblioteki wspomagającej użytkowników w implementacji aplikacji rozproszonych (por. punkt 1.1). Wszystkie przyjęte założenia były weryfikowane poprzez implementację prototypu opisaną w rozdziale 4. W obecnym momencie biblioteka posiada zależność jedynie od usługi Redis wykorzystywanej do realizacji *pamięci rozproszonej*. Należy jednak nadmienić, że zależność tę można w łatwy sposób usunąć poprzez dostarczenie własnej implementacji *pamięci rozproszonej* co jest możliwe dzięki modułowości biblioteki.

Biblioteka i założenia poczynione przy jej tworzeniu zostały dodatkowo zweryfikowane poprzez implementację aplikacji opisanych w rozdziale **??**. Możliwość wykorzystania *debuggera* w trakcie tworzenia tych aplikacji oraz testowania działania biblioteki okazała się niezastąpiona i stanowiła duży atut.

7.2 Napotkane problemy

W trakcie realizacji pracy napotkano problemy, których opis i rozwiązania opisano poniżej. Ich natura była bardzo różna, wynikały głównie z błędów i ograniczeń w wykorzystywanym kodzie pochodzącym od innych twórców:

- serializacja wyjątków,
- niezaimplementowane do końca wewnętrzne metody w Rhino DHT,
- ograniczenie wydajności środowiska Redis w wyniku przenoszenia kodu z systemu Linux do systemu Windows,

- błędy składniowe w plikach XML Schema Definition pochodzących z bibliteki OpenXES.

7.2.1 Serializacja

Wszystkie wyjątki w .NET Framework powinny być serializowalne. Okazuje się jednak, że klasa bazowa `Exception` zawiera wirtualne pole `Data` typu `IDictionary`, które umożliwia dodanie do kolekcji obiektu, który nie będzie serializowalny. Pomimo tego, że każdy wstawiany obiekt jest sprawdzany pod tym kątem poprzez sprawdzenie flagi `Type.IsSerializable`, nie gwarantuje to, że jego składowe będą serializowalne, a tym samym i cały wyjątek.

Drugim problemem jest zachowanie klasy `DataContractSerializer`. Zakładając, że definicja wiadomości (`DataContract`) zawiera pole typu `Exception`, w którym ma zostać przesłany wyjątek, `DataContractSerializer` wymaga dodania atrybutu `KnownType` dla każdego typu wyjątku wywiedzionego z typu `Exception`, który ma zostać przesłany na zdalną stronę, również tego zawartego w polu na wyjątek wewnętrzny (`InnerException`). Dodawanie atrybutu `KnownType` dla każdego typu wyjątku, który może zostać wygenerowany przez kod użytkownika jest rozwiązaniem dalekim od transparentnego, w związku z tym zdecydowano o przesyłaniu wyjątków w formie tekstowej – wywoływana jest na nich rekurencyjnie metoda `ToString`.

7.2.2 Rhino DHT

W punkcie **??**, gdzie opisano wybór aplikacji do zrealizowania *pamięci rozproszonej*, zasygnalizowano problem, który był powodem porzucenia biblioteki Rhino DHT. Zgodnie z założeniami przedstawionymi w punkcie 3.6.1, biblioteka powinna udostępniać mechanizmy atomowego porównania i zapisu bądź atomowego odczytu i zapisu. Przykładowa implementacja *zamka rozproszonego* miała opierać się o mechanizm wersjonowania danych obecny w Rhino DHT:

- pobranie aktualnej, najnowszej wersji pola reprezentującego wartość zamka (wolny/zajęty przez wykonawcę o danym `eid`) wraz z numerem wersji k,
- w przypadku gdy zamek jest wolny – zapis nowej wersji wartości pola (własnego identyfikatora `eid`) ze wskazaniem wersji k jako rodzica,
- odczyt wersji $k+1$ i porównanie z własnym eid – jeżeli jest równe oznacza to, że zamek został pobrany, w przeciwnym wypadku – że został zajęty przez inny proces,
- zwolnienie zamka odbywać się miała poprzez zapis odpowiedniej informacji ze wskazaniem wersji $k+1$ jako rodzica.

Widać zatem, że zapisy miały tworzyć strukturę drzewiastą, gdzie najdłuższa ścieżka miała reprezentować sekwencję pobranych i zwolnionych zamków. Niestety we-

wnętrznie nie wszystkie operacje zostały zaimplementowane, co objawiało się wyjątkiem `NotImplementedException` zgłaszanym z wewnątrz biblioteki Rhino DHT.

7.2.3 Redis uruchamiany w systemie Windows

Podczas pracy usługi Redis z dużym obciążeniem pod kontrolą systemu Windows zaobserwowano znaczący spadek wydajności. Wyjaśnienie problemów z działaniem znaleziono w dokumencie [**?**].

W wersji dla systemu Linux wszystkie operacje wejścia-wyjścia korzystają z deskryptorów plików, które nie są tak rozpowszechnione w API systemu Windows. Wersja przeznaczona dla systemów Windows używa w związku z tym symulowanego deskryptora plików. Zasadniczym problemem jest jednak brak obecności wywołania systemowego `fork` do tworzenia procesów potomnych, która w systemie Windows została zasymulowana poprzez umieszczenie sterty procesu Redis w *memory-mapped file*, czyli fragmencie pamięci wirtualnej, która jest dokładnym odwzorowaniem pliku znajdującego się na dysku lub – w ogólności – obiektu, do którego jesteśmy w stanie uzyskać deskryptor. Taki deskryptor przekazywany jest następnie potomnym procesom. Warto zauważyć, że domyślnym zachowaniem środowiska jest tworzenie współdzielonego pliku o rozmiarze odpowiadającym rozmiarowi pamięci fizycznej komputera.

Ostatecznie różnice, które ujawniły się pomiędzy wersją usługi Redis przeznaczoną dla systemów Windows a pochodnych UNIXa, wpłynęły na podjęcie decyzji o uruchomieniu go w systemie operacyjnym Ubuntu.

7.2.4 Implementacja standardu OpenXES

Podczas próby implementacji standardu OpenXES opisanej w punkcie **??** na podstawie plików XML Schema udostępnionych 28 marca 2014 r. wraz z wersją 2.0 bibliteki napotkano problemy przy próbie automatycznego wygenerowania klas. W jednym z plików, `xes.xsd`, znaleziony został drobny błąd składniowy – brakująca spacja, co powodowało, że plik XML był niepoprawny składniowo i nieprzyjmowany przez narzędzie `xsd.exe`. Po jego poprawieniu ujawnił się kolejny problem objawiający się wyjątkiem przepełnienia stosu w trakcie generowania klas. Ten problem został rozwiązany przez objęcie wnętrza elementu `xs:complexType` odnoszącego się do typu `AttributableType` elementem `xs:sequence`.

7.3 Perspektywy dalszego rozwoju

Przy projektowaniu biblioteki Bluepath został zdefiniowany szereg założeń oraz podjęto szereg decyzji projektowych przedstawionych w rozdziałach trzecim i czwartym.

Poniżej wskazano potencjalne kierunki rozwoju projektu – rozluźniania pewnych założeń i rozbudowy funkcjonalności.

7.3.1 Niezawodność przetwarzania

Aspektem, który nie został poruszony w niniejszej pracy jest zapewnienie niezawodności przetwarzania w przypadku awarii węzłów (por. punkt 3.8). Poniżej zaproponowano sposoby obsługi awarii różnych komponentów – mogą one dotyczyć węzłów danych, węzłów obliczeniowych a także *usługi odnajdywania węzłów*.

Obsługa awarii węzłów danych

Zapewnienie bezpieczeństwa węzłów danych leży po stronie wybranej do ich zrealizowania aplikacji (por. punkt 3.6). Wykorzystywany obecnie system Redis może zostać uruchomiony w trybie z replikacją, w którym jeden z węzłów jest punktem dostępowym do zapisu lub odczytu, a jego repliki udostępniają dane tylko do odczytu. Po awarii węzła nadrzędnego jedna z replik staje się nowym punktem dostępowym z możliwością zapisu.

Istnieje również wariant systemu – Redis Cluster – w którym dane dzielone są pomiędzy węzły, udostępniający mechanizmy replikacji i rekonfiguracji w przypadku awarii.

Obsługa awarii węzłów obliczeniowych

W przypadku, gdy aplikacja nie posiada współdzielonego stanu, czyli nie wykorzystuje rozproszonych struktur danych ani obiektów – a więc wywołania są idempotentne – obsługa awarii węzłów obliczeniowych mogłaby uwzględniać drzewiasty charakter takiego przetwarzania. *Wątki rozproszone* zainicjowane przez stracony węzeł musiałyby być wycofane, a węzeł nadrzędny musiałby zlecić jeszcze raz wykonanie przerwanego wątku. *Wątki rozproszone*, które mogą być bezpiecznie uruchamiane ponownie musiałyby być *explicite* oznaczane przez programistę flagą (`Resumable`) lub z wykorzystaniem dedykowanej metody (`AsResumable`).

Obsługa awarii serwera usługi odnajdywania węzłów

Wrażliwym punktem jest również scentralizowany serwer *usługi odnajdywania węzłów*. Warto jednak zauważyć, że prawdopodobieństwo awarii jednego konkretnego węzła jest dużo niższe niż prawdopodobieństwo awarii dowolnego węzła w klastrze. Argumentem, który przemawia za koniecznością zajęcia się tym problemem jest kluczowe znaczenie serwera *usługi odnajdywania węzłów* do realizacji przetwarzania. Aby zapewnić niezawodność dostępu do *usługi odnajdywania węzłów* możnaby uruchomić jego replikę, która przejęłaby zadania po awarii podstawowego serwera

(podejście to wykorzystywane jest np. w MapReduce, gdzie *name node* jest replikowany). Warto również rozważyć implementację opartą o rozproszoną tablcę haszową (ang. *distributed hash table*, DHT) z replikacją.

7.3.2 Rozbudowa funkcjonalności

Ponieważ z przetwarzaniem rozproszonym wiąże się wiele problemów, a przy tworzeniu biblioteki zauważano kolejne obszary w których możnaby zmniejszyć nakład pracy użytkownika nie było możliwe zrealizowanie wszystkich funkcji. Poniżej przedstawiono przykładowe obszary, w których możnaby rozbudować funkcjonalność systemu:

- rozproszone struktury danych – np. implementacja kolejki,
- interfejs do komunikacji z systemem – rozbudowa o metody dostępu do pamięci lokalnej węzła (`LocalStorage`), dodanie możliwości raportowania zaawansowania przetwarzania w ramach wątku (`ReportProgress`) w celu budowy dokładniejszych planistów,
- DLINQ – implementacja pozostałych metod obecnych w standardowym LINQ i niedostępnych w obecnej implementacji,
- wykorzystanie asynchronicznych wzorców z .NET Framework 4.5 (`async`, `await`) – obecnie jedyną operacją asynchroniczną jest oczekiwanie na zakończenie *wątku rozproszonego* – `JoinAsync`.

7.4 Zakończenie

W niniejszej pracy przeanalizowano i porównano istniejące rozwiązania wspomagające programistów w tworzeniu aplikacji rozproszonych oraz zaprojektowano i zaimplementowano prototyp biblioteki, która pozwala jej użytkownikom w prosty sposób zaimplementować program przetwarzający dane w sposób rozproszony. Użytkownik pisząc program w sposób podobny do tego, w jaki pisałby program równoległy, uzyskuje program wykorzystujący moc obliczeniową wielu węzłów w klastrze. W takim przypadku wątki, zamiast wykonywać się na wielu rdzeniach jednego procesora, wykonują się na zdalnych maszynach. Prototyp biblioteki został przetestowany zarówno pod względem jakościowym jak i wydajnościowym. Przedstawiono również przykładowe zastosowania biblioteki. Biorąc powyższe pod uwagę można uznać, że wszystkie założone cele pracy zostały zrealizowane.