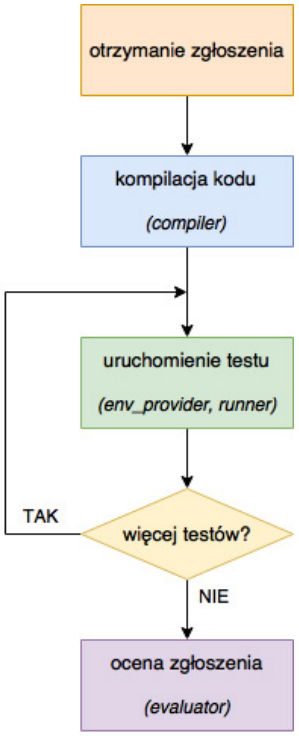
Projekt Algochecker – skalowalna platforma na bazie Dockera do automatycznego testowania programów

Projektowanie i implementacja platformy przeznaczonej do automatycznego oceniania wydajności algorytmów nie jest zadaniem trywialnym. Rozwiązania tego typu klasyfikuje się jako „online judge system", czyli w dosłownym tłumaczeniu „sędzia internetowy". Różnią się one zbiorem obsługiwanych języków, możliwościami oraz sposobem implementacji. W tym artykule zaprezentujemy architekturę naszego rozwiązania, podstawowe przypadki użycia i wyzwania inżynierskie, które za nim stoją.

JAK DZIAŁA „SĘDZIA INTERNETOWY"?

Zacznijmy od scharakteryzowania wspomnianych „sprawdzarek". Z punktu widzenia użytkownika schemat blokowy takiego rozwiązania zazwyczaj wygląda następująco:



Rysunek 1. Typowy schemat blokowy procesu testowania zgłoszeń. W nawiasach podano nazwy abstrakcji powiązanych z danymi etapami w systemie Algochecker

Przykładami istniejących usług działających w podobny sposób są Sphere Online Judge (SPOJ) [1], HackerRank [2], UVa Online Judge [3] czy Codility [4]. We wszystkich wymienionych przypadkach dostęp użytkownika (nazywanego dalej uczestnikiem) odbywa się z poziomu aplikacji webowej, która zawiera listę udostępnionych mu zadań programistycznych i umożliwia wrzucenie swojego projektu w postaci plików z kodem źródłowym w danym języku programowania.

Wysłanie plików z kodem skutkuje ich kompilacją (w przypadku języków kompilowanych) i uruchomieniem w izolowanym środowisku. Jeżeli wynikowy program spełni postawione przed nim wymagania, zostanie oceniony pozytywnie.

Powszechnie stosowany, najbardziej intuicyjny i zarazem najprostszy sposób zdefiniowania takich wymagań to przygotowanie kilku wzorcowych plików wejściowych i wyjściowych. Program napisany przez uczestnika, po przekazaniu do niego wzorcowego pliku wejściowego, powinien:

- obliczyć wynik, mieszcząc się w określonym limicie czasu (zazwyczaj wartości rzędu 1 sekundy),
- prawidłowo zakończyć pracę, zwracając kod wyjścia (ang. *exit code*) 0,
- wydrukować wynik zgodny z wzorcowym plikiem wyjściowym, przygotowanym przez autora zadania.

Jest to najbardziej rozpowszechniony sposób testowania, jednak jest on obarczony sporymi ograniczeniami. Nie jest możliwe testowanie w ten sposób rozwiązań problemów, w których nie wszystkie dane wejściowe są znane od początku. Możliwe jest więc zdefiniowanie zadania: „napisz program, który dla danego grafu skrzyżowań w mieście znajdzie najkrótszą ścieżkę", przy założeniu, że program od początku swojego działania posiada wiedzę na temat całej sieci dróg. Nie ma jednak możliwości przetestowania w ten sposób chociażby algorytmu AI do gry w szachy, gdzie na starcie nie posiadamy informacji o wszystkich przyszłych ruchach naszego przeciwnika.

Kwestia bardziej skomplikowanych sposobów testowania została opisana w dalszej części artykułu. To wszystko, jeśli chodzi o wprowadzenie teoretyczne, pozostałą część materiału poświęcimy omówieniu naszej, konkretnej implementacji tego zagadnienia wraz ze wskazaniem właściwości takiego rozwiązania.

STRUKTURA SYSTEMU ALGOCHECKER

Nasze rozwiązanie składa się z czterech zasadniczych bloków funkcjonalnych:

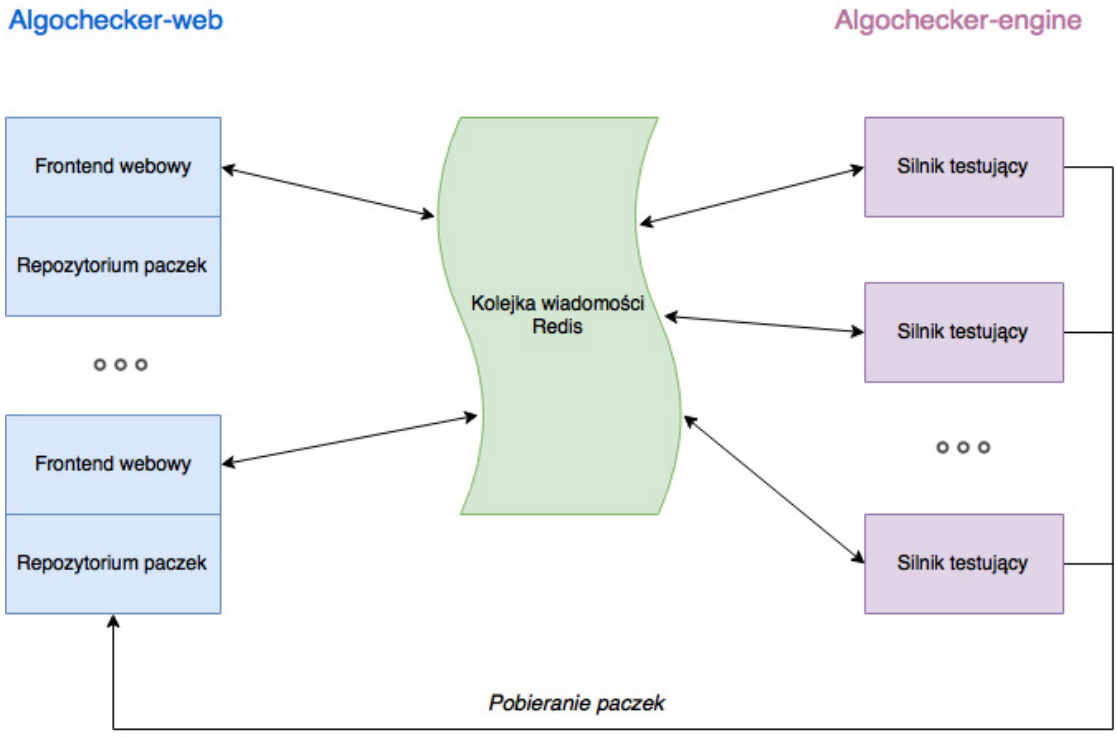
Frontend webowy – podprojekt algochecker-web odpowiedzialny jest za „obsługę klienta", czyli autoryzację użytkowników, prezentację treści zadań, wysyłkę nowych zgłoszeń do sprawdze-

Sphere online judge

judge status

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
19372847	2017-05-09 17:57:58	dian	Prime Generator	compilation error	-	-	JAVA
19372845	2017-05-09 17:57:54	Annan	Character Patterns (Act 3)	**accepted**	0.00	2.7M	C++ 4.3.2
19372844	2017-05-09 17:57:46	gokul	Alphacode	wrong answer	0.01	3.3M	C++ 4.3.2
19372841	2017-05-09 17:57:21	gokul	Alphacode	wrong answer	0.01	3.3M	C++ 4.3.2
19372840	2017-05-09 17:57:00	AAA	Fashion Shows	**accepted**	0.01	16M	CPP14
19372838	2017-05-09 17:56:47	AAA	Fashion Shows	**accepted**	0.00	15M	CPP14
19372836	2017-05-09 17:55:44	hieuvt3112	Easy sudoku	time limit exceeded	-	4284M	JAVA
19372835	2017-05-09 17:55:31	anaar_13	Odd Numbers of Divisors	wrong answer	0.01	17M	CPP14-CLANG
19372831	2017-05-09 17:55:00	gokul	Divisor Summation	**accepted**	0.36	15M	CPP
19372830	2017-05-09 17:54:53	Ayushi	Half of the half	wrong answer	0.00	9.2M	C
19372829	2017-05-09 17:54:50	Navo	Prime Generator	wrong answer	0.01	9.2M	C

Rysunek 2. Przykładowa tabela wyników z serwisu spoj.com, prezentująca wyniki oceny zgłoszeń ostatnio nadesłanych przez uczestników. W kolumnie „problem" podana jest nazwa zadania programistycznego, w kolumnie „result" – wynik automatycznych testów



Rysunek 3. Podstawowe bloki funkcjonalne w systemie Algochecker

nia oraz prezentację wyników dla uczestników i administratorów. Komunikacja z częścią oceniającą zgłoszenia odbywa się za pośrednictwem kolejki wiadomości Redis, poprzez narzucony protokół, który musi być przestrzegany w obrębie całego systemu.

System decyzyjny odpowiedzialny za sprawdzanie zgłoszeń znajduje się w pod-projekcie algochecker-engine (pojedynczą instancję tego programu nazywamy „worker"). Jego zadania to: pobranie zgłoszenia z kolejki wiadomości, skompilowanie znajdującego się w nim kodu źródłowego, uruchomienie wynikowego programu w izolowanym środowisku, przetestowanie zgodnie z ustalonymi kryteriami i wygenerowanie raportu końcowego.

Konfiguracja poszczególnego zadania ma postać archiwum .zip i jest nazywana potocznie „paczką". Repozytorium paczek jest zintegrowane w aplikacji webowej, ale może być również niezależnym, małym serwerem HTTP. Żeby rozpocząć ocenianie danego zgłoszenia, silnik musi otrzymać nazwę zadania, wersję (timestamp), adres URL paczki konfiguracyjnej oraz kod źródłowy stworzony przez uczestnika.

Komunikacja z pozostałymi częściami systemu również tym razem odbywa się za pośrednictwem kolejki wiadomości Redis z wykorzystaniem mechanizmów BLPOP [5] (pobieranie zgłoszeń z kolejki) oraz PUBSUB [6] (rozgłaszanie wyników). Za wszystkie pozostałe czynności, tzn. wrzucanie zgłoszeń do kolejki, zapisywanie ogłoszonych wyników w bazie danych itp., odpowiada aplikacja webowa.

Jedną z możliwych form organizacji danych w Redisie jest kolejka (ang. *queue*), do której możliwe jest dodawanie elementów na lewy lub prawy kraniec. Analogicznie możliwe jest wyciąganie elementów z danego krańca w sposób blokujący albo nieblokujący. W Algocheckerze nowe zgłoszenia dodawane są do kolejki poprzez polecenie RPUSH (zakolejkowanie na prawym krańcu), a pobierane przez BLPOP (wyciągnięcie elementu z lewego krańca, jeżeli kolejka jest pusta – blokowanie do momentu, aż ten stan rzeczy ulegnie zmianie).

Inny dostępny sposób przetwarzania danych to Pub/Sub, czyli mechanizm kanałów, do których możliwe jest nadawanie wiadomości oraz ich subskrybowanie. Zarówno nadawców, jak i subskrybentów może być wielu.

Dzięki tym rozwiązaniom możliwe jest skalowanie systemu poprzez dodawanie dodatkowych instancji algochecker-engine oraz algochecker-web, działających w ramach jednej kolejki wiadomości.

BUDOWA SILNIKA TESTUJĄCEGO

Serce projektu, czyli projekt „engine", składa się wewnętrznie z kilku luźnych modułów Pythona zapewniających abstrakcję nad Dockerem i Redisem, metod umożliwiających wykonywanie koniecznych health-checków oraz głównej pętli, która steruje uruchamianiem kolejnych etapów testowania.

Proces testowania podzielony jest na trzy logiczne części:

- kompilacja – wykonywana raz, na początku przetwarzania zgłoszenia,
- testowanie – dla każdego zdefiniowanego testu, przygotowanie odpowiedniego środowiska i uruchomienie programu,
- ocena – końcowa interpretacja wyników testowania.

Na potrzeby kompilacji i testowania programów silnik Algocheckera za każdym razem tworzy nowy kontener Dockera przeznaczony do wykonania pojedynczej czynności, po czym kontener jest niszczony.

PIPELINE PLUGINÓW

Kluczowymi elementami całego procesu oceniania zgłoszeń są cztery rodzaje pluginów, które pracują zgodnie z kolejnością przedstawioną na Rysunku 1:

- `compiler` – plugin obsługujący określony kompilator (np. `gcc`), danymi wejściowymi są kod źródłowy oraz konfiguracja, a rezultatem działania skompilowany program oraz opcjonalnie logi z procesu kompilacji (ostrzeżenia kompilatora, błędy etc.),
- `env_provider` – plugin zajmujący się przygotowaniem środowiska do przeprowadzenia konkretnego testu i wystawieniem oceny cząstkowej za test,
- `runner` – plugin uruchamiający niezaufany kod (program użytkownika) w izolowanym środowisku z konfiguracją narzuconą przez env_providera,
- `evaluator` – plugin obliczający ocenę końcową na podstawie szczątkowych wyników testowania.

Przedstawiony podział umożliwia rozszerzanie możliwości systemu stosunkowo niskim kosztem. Na przykład, działający pipeline do testowania zgłoszeń w języku C można przestawić na dowolny inny kompilowany język programowania, dopisując zaledwie jeden plugin – `compiler`.

W przypadku języków interpretowanych (np. Python, PHP) etap kompilacji jest pusty lub sprowadza się wyłącznie do sprawdzenia poprawności składni, a poprawne uruchomienie skryptu staje się odpowiedzialnością pluginu typu `runner`.

Kryteriami oceniania zgłoszeń można manipulować, zmieniając konfigurację pluginu `evaluator` lub tworząc własny plugin tego typu. Przykładem takiego działania może być modyfikacja punktacji (przydzielenie wyższych/niższych wag punktowych poszczególnym testom) lub wprowadzenie powiązań pomiędzy testami (np. niezaliczenie testu numer 3 powoduje automatyczne wyzerowanie wyników za testy numer 4 i 5).

Każdy z tych pluginów może w razie potrzeby stworzyć jeden lub więcej kontenerów Dockera przeznaczonych do przeprowadzenia określonego etapu testowania. Zapewnione pluginom pomocnicze abstrakcje są odpowiedzialne za uniemożliwienie globalnego wycieku zasobów w przypadku błędnego działania pluginu (spowodowanego np. błędną implementacją lub nieobsłużonym wyjątkiem).

PROSTY PRZYKŁAD TESTOWANIA

Pełny kod zamieszczonych przykładów paczek konfiguracyjnych dostępny jest w repozytorium projektu (odnośnik znajduje się w przypisach na końcu artykułu), a uruchomić go można, wykorzystując demonstracyjną maszynę wirtualną z preinstalowanymi narzędziami (sekcja „Spróbuj i Ty!").

Najprostszym sposobem testowania jest wspomniane na początku artykułu wykorzystanie pary plików, za pomocą których autor zadania zapewnia dane wejściowe, oraz wzorcowy plik wynikowy. Taką logikę realizuje plugin typu `env_provider` o nazwie `file`.

Za przykład weźmy zadanie: „napisz program w języku C, który posortuje N liczb z wejścia standardowego i wypisze je na wyjściu standardowym". Konfiguracja paczki służącej do przetestowania rozwiązań takiego zadania będzie wyglądała następująco:

Listing 1. Przykładowa konfiguracja paczki z zadaniem, plik config.yml

```
configs:
  _base:
    compiler:
      name: "gcc"
      command_line: "-Wall -O2 -std=c++11"
      limits:
        timeout: 30000
        cpu_quota: 25000
        cpu_period: 50000
        max_memory: "32M"
    env:
      name: "file"
    runner:
      name: "bin"
      limits:
      timeout: 1000
      cpu_quota: 25000
      cpu_period: 50000
      max_memory: "32M"
    evaluator:
      name: "basic"
```

Za każdym razem, podczas definiowania zadania, układamy określony „pipeline" pluginów. W tym celu konieczne jest wymienienie ich nazw w konfiguracji, w tym przykładzie: plugin kompilujący – `gcc`, plugin środowiska testowego – `file` (testowanie oparte o pliki wejściowe/wyjściowe), plugin uruchamiający – `bin` (generyczny do programów w postaci skompilowanego ELF), oceniacz – `basic` (standardowy).

Użyte w przykładzie opcje konfiguracyjne są raczej dosyć intuicyjne, z zastrzeżeniem, że `timeout` [ms] to limit czasu wykonania się określonego etapu (30 sekund na kompilację, 1 sekunda na wykonanie pojedynczego testu), `cpu_quota` [µs] to maksymalny czas wykorzystywania procesora w okresie o długości `cpu_period` [µs]. Maksymalne dozwolone wykorzystanie rdzenia procesora w czasie działania programu można oszacować, używając wzoru: `100 * cpu_quota / cpu_period` [%].

Po utworzeniu pliku konfiguracyjnego konieczne jest jeszcze opracowanie przynajmniej kilku przypadków testowych. Jakie przypadki testowe najlepiej wybrać? Kilka prostych, łatwych do zweryfikowania, kilka przypadków brzegowych – nietypowych, których użycie mogłoby spowodować nieprawidłowe działanie algorytmu, i kilka nastawionych na sprawdzenie wydajności.

TESTOWANIE ZACHOWANIA PROGRAMU DLA TYPOWEGO WEJŚCIA (TESTY TRYWIALNE)

Przykładowe zadanie polega na napisaniu programu sortującego N liczb, wybieramy stosunkowo małe N (np. 4) i tworzymy przypadek testowy w oparciu o losowe dane.

Listing 2. Przykładowy test trywialny

Plik: input/test1-small.txt

```
4
17
9
20
1
```

Plik: output/test1-small.txt

```
1
9
17
20
```

Nazwy plików z testami (tutaj: `test1-small.txt`) są bez znaczenia, z zastrzeżeniem, że będą one wyświetlane użytkownikom w raporcie końcowym.

Testy przypadków brzegowych

Jak zachowa się program, jeżeli podamy tylko jedną liczbę? Co, jeżeli podamy 100 identycznych liczb? Co, jeśli treść zadania dopuszcza N=0 i nie podamy żadnej liczby?

Listing 3. Przykładowy test przypadku brzegowego

Plik: input/test2-empty.txt

```
0
```

Plik: output/test2-empty.txt

(pusty)

Listing 4. Inny przykład przypadku brzegowego

Plik: input/test3-single-number.txt

```
3
1
1
1
```

Plik: output/test3-single-number.txt

```
1
1
1
```

Testy wydajnościowe

Algorytm o akceptowalnej złożoności powinien być w stanie posortować zbiór 10000 liczb, nie wymagając przy tym wiele czasu. Poprzez tworzenie testów z dużymi zbiorami danych i ustawianie niskich limitów czasu oraz niskich przydziałów CPU możliwe jest eksperymentalne zidentyfikowanie złożoności obliczeniowej nadesłanego rozwiązania. Najczęstszą praktyką jest tworzenie testów, które nie zaakceptują określonych klas złożoności, np. $O(n^2)$. W tym przypadku ustawienia muszą być wyskalowane z uwzględnieniem faktycznej konfiguracji sprzętowej serwera testującego.

> Konfiguracja i wdrożenie platformy Algochecker, proces tworzenia zadań oraz instalowania paczek są opisane w dokumentacji przeznaczonej dla administratorów. Artykuł intencjonalnie pomija tę tematykę, ponieważ wspomniane czynności są trywialne dla każdego administratora systemów.

Kiedy nasza paczka konfiguracyjna jest już gotowa, możliwe jest zainstalowanie i sprawdzenie jej działania poprzez wysłanie kilku testowych rozwiązań stworzonego zadania.

Uruchomienie testów

Listing 5. Poprawne rozwiązanie problemu sortowania zbioru liczb

```
#include <iostream>
#include <random>
#include <algorithm>
```

```cpp
using namespace std;

int main()
{
  int N;
  cin >> N;

  std::vector<int> numbers;

  for (int i = 0; i < N; i++) {
    int temp;
    cin >> temp;
    numbers.push_back(temp);
  }

  std::sort(numbers.begin(), numbers.end());

  for (std::vector<int>::iterator it=numbers.begin(); it !=
  numbers.end(); ++it) {
    cout << *it << endl;
  }

  return 0;
}
```

Tests

Name	Status	Points received/max	Time ms	Memory MB
test1-small	ok	1.0 / 1.0	3	1.098
test2-empty	ok	1.0 / 1.0	5	1.066
test3-single-number	ok	1.0 / 1.0	4	1.094
test4-big-set	ok	1.0 / 1.0	30	1.238

Rysunek 4. Wynik oceny poprawnego rozwiązania do zadania o sortowaniu zbioru liczb

Dokonajmy teraz podmiany linijki odpowiedzialnej za sortowanie w naszym kodzie:

Listing 6. Zmiana kodu rozwiązania

```cpp
std::sort(numbers.begin(), numbers.end());
// zmieniamy na:
std::random_shuffle(numbers.begin(), numbers.end());
```

Tym razem nasze zgłoszenie poprawnie przechodzi jedynie przez dwa z czterech przygotowanych testów. Dzieje się tak, ponieważ pomimo użycia funkcji `random_shuffle` końcowy wynik jest poprawny dla danych wejściowych w postaci pustego zbioru albo zbioru z pojedynczą liczbą powtórzoną kilkukrotnie.

Tests

Name	Status	Points received/max	Time ms	Memory MB
test1-small	bad answer	0.0 / 1.0	2	1.121
test2-empty	ok	1.0 / 1.0	2	1.133
test3-single-number	ok	1.0 / 1.0	2	1.066
test4-big-set	bad answer	0.0 / 1.0	26	1.234

Rysunek 5. Wynik oceny niepoprawnego rozwiązania wykorzystującego random_shuffle

ROZSZERZONE MOŻLIWOŚCI TESTOWANIA (SERWISY)

Praktycznie każdy system typu *online judge* jest w stanie przetestować opisany wyżej przykładowy problem. Pora na przedstawienie bardziej nietypowego przykładu, dalej zaprezentowany zostanie proces definiowania następującego zadania:

„Napisz program, który będzie zgadywał losowo wygenerowaną liczbę od 0 do 1000000. Program powinien wypisać propozycję liczby na swoje wyjście standardowe, po czym na swoje wejście standardowe otrzyma jedną z wartości:

- `-1` zaproponowana liczba jest mniejsza od wylosowanej,
- `1` zaproponowana liczba jest większa od wylosowanej,
- `0` prawidłowe trafienie, program może zakończyć pracę.

Opisany protokół należy powtarzać w pętli, aż do odgadnięcia prawidłowej liczby (otrzymania zwrotnej wartości 0). Liczba musi zostać odgadnięta w ciągu maksymalnie 20 iteracji".

Podany problem wymaga interakcji dwóch programów, które zrealizują powyższy protokół. Tym razem – zamiast używać wzorcowych plików wejścia/wyjścia – stworzymy własny skrypt (nazywany serwisem), który będzie komunikował się z testowanym programem użytkownika.

Listing 7. Przykładowa konfiguracja, plik config.yml

```yaml
configs:
  _base:
    compiler:
      name: "gcc"
      command_line: "-Wall -O2 -std=c++11"
      limits:
        timeout: 30000
        cpu_quota: 25000
        cpu_period: 50000
        max_memory: "32M"
    env:
      name: "pipe"
    service_runner:
      name: "bin"
      image: "python:latest"
      limits:
        timeout: 30000
        cpu_quota: 25000
        cpu_period: 50000
        max_memory: "256M"
    runner:
      name: "bin"
      limits:
        timeout: 1000
        cpu_quota: 25000
        cpu_period: 50000
        max_memory: "32M"
    evaluator:
      name: "basic"
```

Przedstawiona konfiguracja różni się od poprzedniej zawartością sekcji `env` – tym razem wybieramy plugin, który przeprowadza testowanie poprzez „spięcie" dwóch programów potokami (ang. *pipe*). Program napisany przez użytkownika będzie miał przekierowane strumienie stdin i stdout, widziane przez serwis w systemie plików jako dwa pliki specjalne, które może czytać i zapisywać. Ponownie: testów zostanie wykonanych tyle, ile plików utworzymy w katalogu `input/` naszej paczki konfiguracyjnej. Plugin `pipe` zmienia jednak ich znaczenie. Tym razem są one po prostu ustawieniami serwisu bez ściśle zdefiniowanego formatu.

Przedstawiona konfiguracja, w odróżnieniu od poprzedniej, zawiera dwie sekcje „uruchamiacza": `runner` i `service_runner`. Stało się tak, ponieważ tym razem do przeprowadzenia pojedynczego testu będą konieczne dwa kontenery. Pierwsza sekcja opisuje więc środowisko uruchomieniowe, w którym będzie testowany program zapewniony przez użytkownika, a druga środowisko przeznaczone dla skryptu serwisu stworzonego przez autora zadania.

Po stworzeniu pliku konfiguracyjnego następnym krokiem jest napisanie skryptu serwisu. W przykładzie posłużymy się Pythonem, choć możliwe jest wykorzystanie każdej technologii, do której istnieje gotowy obraz Dockera (z definicji możliwe jest również sforkowanie istniejącego obrazu i dostosowanie go do swoich potrzeb). Skrypt należy zaimplementować przy uwzględnieniu następujących reguł:

- Potok umożliwiający pisanie na wejście standardowe programu użytkownika znajduje się w `/mnt/prog-in/input.txt`.
- Potok umożliwiający czytanie tego, co wypisał program użytkownika, znajduje się w `/mnt/prog-out/output.txt`.
- Na swoje wejście standardowe skrypt otrzymuje konfigurację: treść jednego z plików z katalogu `input/`.
- Jeżeli test zakończył się pomyślnie, skrypt powinien wypisać dane w formacie JSON zawierające pola: `points` (punkty przyznane za test), `max_points` (maksymalna liczba punktów do uzyskania w tym teście) i opcjonalnie `message` (dodatkowa wiadomość diagnostyczna).

Listing 8. Skrypt serwisu, z którym będzie komunikował się testowany program

```
#!/usr/local/bin/python
import json
import sys
import errno

def broken_pipe_error():
    print(json.dumps({'points': 0, 'max_points': 1, 'message':
'The program terminated unexpectedly.'}))
    sys.stdout.flush()
    exit(0)

def main():
    try:
        fin = open('/mnt/prog-in/input.txt', 'w')
        fout = open('/mnt/prog-out/output.txt', 'r')

        # give a signal that we are ready to start the test
        # by unlocking a file spinlock in the engine
        open('/mnt/out/srv-ready', 'w').close()

        # read the number to be guessed from input/* file
provided at stdin
        actual_number = int(sys.stdin.readline().strip())
        user_guess = None
        num_guess = 0

        while actual_number != user_guess:
            try:
                user_guess = int(fout.readline().strip())
            except ValueError:
                print(json.dumps({"points": 0.0, "max_points":
1.0, "message": "Invalid value provided by the program."}))
                sys.stdout.flush()
                exit(0)

            num_guess += 1
            if user_guess > actual_number:
                fin.write("1\n")
            elif user_guess < actual_number:
                fin.write("-1\n")
            else:
                fin.write("0\n")

            fin.flush()

        if num_guess <= 20:
            points = 1.0
        else:
            points = max(0, 1.0 - (num_guess - 20) / 10)

        print(json.dumps({"points": points, "max_points": 1.0,
"message": ""}))
        sys.stdout.flush()
    except IOError as e:
        if e.errno == errno.EPIPE:
            broken_pipe_error()
        else:
            raise e

main()
```

W tym momencie konfiguracja jest gotowa, a po jej załadowaniu możemy ponownie wcielić się w rolę uczestnika i napisać pierwszy kod rozwiązujący problem zgodnie z ustalonym wcześniej protokołem.

Listing 9. Kod programu, który próbuje odgadnąć liczbę poprzez losowanie

```
#include <iostream>
#include <random>

using namespace std;

int main()
{
  int status;

  do {
    cout << rand() % 1000001 << endl;
    cin >> status;
  } while (status);

  return 0;
}
```

Nie jest to najlepsze rozwiązanie, w związku z tym nie dziwi następujący wynik oceny przez Algocheckera:

Tests

Name	Status	Points received/max	Time ms	Memory MB
test1	hard timeout	0.0 / 1.0	≥ 1500	n/a
test2	hard timeout	0.0 / 1.0	≥ 1500	n/a

Rysunek 6. Wynik oceny niepoprawnego rozwiązania – program nie zdążył wylosować prawidłowej liczby przed upływem zadanego limitu czasu

Listing 10. Poprawny kod

```
#include <iostream>

using namespace std;

int main()
{
  int low_bound = 0;
  int high_bound = 1000000;
  int guess = (high_bound + low_bound) / 2;
  int status;

  do {
    cout << guess << endl;
    cin >> status;

    if (status == 1) {
      // guess > actual_number
      high_bound = guess;
    } else if (status == -1) {
      // guess < actual_number
      low_bound = guess;
    }

    guess = (high_bound + low_bound) / 2;
  } while (status != 0);

  return 0;
}
```

Tests

Name	Status	Points received/max	Time ms	Memory MB
test1	ok	1.0 / 1.0	5	1.105
test2	ok	1.0 / 1.0	9	1.176

Rysunek 7. Wynik oceny poprawnego kodu, wykorzystującego technikę „dziel i zwyciężaj”

PROGRAMISTYCZNE „SMACZKI”

Podczas powyższej prezentacji niektórych funkcji pojawiły się sztuczki dosyć specyficzne dla tej implementacji. Poniżej scharakteryzujemy kilka specyficznych problemów inżynierskich napotka-

nych podczas tworzenia platformy w oparciu o Dockera oraz ich rozwiązania zastosowane w projekcie Algochecker.

Różnice pomiędzy command line i API Dockera

Podczas projektowania oprogramowania, które automatycznie wywołuje Dockera, zaskoczenie mogą wywołać różnice pomiędzy interfejsem command line i dostępnym API. Udostępniane programistom bindingi operują na nieco niższym poziomie abstrakcji niż komendy i cechują się inną konwencją nazewnictwa.

W pierwszej wersji API Dockera 1.10 dolegliwe były różnice pomiędzy nazwami komend i metod, na przykład: wywołaniem równoważnym do `docker rm {}` jest tam `docker.remove_container(container)`. Znacznie większą niespodzianką był brak odpowiednika polecenia `docker run`, co wymuszało tworzenie kilku pośrednich obiektów ręcznie (polecenia: `create_container`, `create_host_config`, `start`, `wait`, `stop`).

W trakcie pisania artykułu wspomniane problemy zostały naprawione w docker-py 2.0, które przeszło refaktoryzację i jest teraz bardziej zorientowane obiektowo oraz zawiera metody wysokopoziomowe takie jak `container.run()`. Niemniej jednak, wspomniane „zwięzłe" metody miałyby znaczenie jedynie w bardzo początkowej fazie projektu – ich możliwości zazwyczaj szybko się wyczerpują, co generuje konieczność rozbicia jednego wywołania API na kilka bardziej niskopoziomowych.

Szybkie statystyki kontenera

Wywołanie API `docker.stats(container, decode=True, stream=False)` [7] z nieokreślonych przyczyn zajmuje nawet do trzech sekund. Oczekiwanie na wynik działania tej metody okazał się główną składową całego czasu testowania. Zamiast marnować czas, można wyciągnąć niektóre statystyki bezpośrednio:

- `/sys/fs/cgroup/memory/docker/{container_id}/memory.max_usage_in_bytes` – maksymalne zużycie pamięci kontenera,
- `/var/lib/docker/containers/{container_id}/config.v2.json` – konfiguracja kontenera, również kod błędu, czas startu i zakończenia pracy z ostatniego uruchomienia.

Peak memory usage

Pomiar maksymalnego zużycia pamięci można łatwo zrealizować poprzez wirtualny plik udostępniany przez cgroups, wspomniany w poprzednim punkcie. W zależności od obrazu Dockera kontener na samym początku swojego działania może osiągnąć peak memory usage na poziomie 10-20 MB. Powoduje to niepożądane zaburzenie dokładności pomiaru (wynik będzie wiarygodny, dopiero jeżeli testowany program sam zajmie więcej niż 20 MB pamięci).

Możliwe jest jednak nadpisanie `memory.max_usage_in_bytes` arbitralną wartością, np. w ten sposób:

```
echo -n 0 > /sys/fs/cgroup/memory/docker/{container_id}/memory.max_usage_in_bytes
```

Jeżeli zostanie to zrobione po starcie kontenera, ale przed startem testowanego programu, pomiar maksymalnego zużycia pamięci stanie się znacznie bardziej wiarygodny. Ponieważ cgroups są zapewniane przez sam kernel Linuxa, to rozwiązanie prawdopodobnie jest przenośne pomiędzy dystrybucjami (chociaż nie ma takiej gwarancji).

File spinlock [8]

Niekiedy istnieje konieczność synchronizacji pomiędzy testowanym programem w kontenerze a workerem (np. żeby precyzyjnie zmierzyć czas od faktycznego rozpoczęcia testu do jego zakończenia). Programy nie znajdujące się w kontenerze są z definicji całkowicie niewidoczne wewnątrz niego, co uniemożliwia komunikację z użyciem sygnałów lub IPC (bez mapowania pamięci).

Jednym z najprostszych i zarazem najbardziej pewnych sposobów realizacji tego zagadnienia jest oczekiwanie na utworzenie pliku o ustalonej nazwie. Metoda ta może wydawać się nieco kontrowersyjna, ale jej niewątpliwą zaletą jest bardzo wysoka kompatybilność, niska podatność na błędy i prostota debugowania – właściwości pożądane podczas eksperymentów z nowym oprogramowaniem.

Ideę mogą wyjaśnić dwa proste skrypty w bashu.

Listing 11. Atrapa skryptu a.sh

```
#!/bin/bash

echo "Starting A..."
sleep 5
echo "A is done"
touch a_done
```

Listing 12. Skrypt b.sh, który musi wykonać się po skrypcie a.sh

```
#!/bin/bash

while [ ! -f ./a_done ]
do
  sleep 0.1
fi

echo "Starting B..."
sleep 2
echo "B is done..."
```

Podany przykład można uruchomić np. w ten sposób: `rm -f a_done && ./a.sh | ./b.sh`. Spowoduje to usunięcie pliku o nazwie `a_done` (o ile takowy istnieje), a później równoległe uruchomienie skryptów `a.sh` i `b.sh`.

Powyższy skrypt zamiast pętli `while` zawierającej `sleep` mógłby alternatywnie wykorzystać polecenie `inotifywait`, które blokowałoby do momentu otrzymania zdarzenia o utworzeniu pliku.

Named pipe pomiędzy kontenerami

W przypadku kiedy proces testowania wymaga skomunikowania programów znajdujących się w dwóch różnych kontenerach Dockera. Za pomocą komendy `mkfifo` możliwe jest utworzenie tzw. *named pipe* [9], czyli pliku typu *device* obsługiwanego w specjalny sposób.

Nawiązanie komunikacji w ten sposób musi być poprzedzone utworzeniem *named pipe* (nazwanego potoku) oraz otwarciem go przez jeden program w trybie zapisu, a przez drugi w trybie odczytu. Zapisanie danych do potoku spowoduje, że trafią one bezpośrednio do bufora potoku z pominięciem faktycznego systemu plików. Działanie to jest analogiczne do klasycznego przekierowania za pomocą potoku nienazwanego (tzn. `./program_a | ./program_b`).

BEZPIECZEŃSTWO

Prezentowane rozwiązanie zapewnia izolację procesów w dużej mierze opartą na mechanizmie cgroups, wewnętrznie stosowa-

nym w Dockerze. W przeciwieństwie do maszyny wirtualnej jądro jest współdzielone pomiędzy „prawdziwym systemem" i kontenerami [10].

W związku z tym potencjalnym zagrożeniem mógłby być exploit na kernel, który byłby w stanie zadziałać wewnątrz nieuprzywilejowanego kontenera. Z tego powodu dla zapewnienia bezpieczeństwa kluczowe jest posiadanie na serwerach aktualnego kernela z nowoczesnej, wspieranej dystrybucji. Prawdopodobieństwo wystąpienia takiego ataku jest jednak stosunkowo niewielkie.

Można zastosować dodatkowe zabezpieczenia, takie jak np. rozszerzenie grsec. Możliwe jest również umieszczenie całego systemu wewnątrz maszyny wirtualnej, aby dodatkowo zabezpieczyć się przed potencjalną eskalacją ataku. Poziom nałożonych zabezpieczeń w poszczególnym wdrożeniu powinien być uzależniony od indywidualnej kalkulacji ryzyka.

SPRÓBUJ I TY!

Na stronie projektu dostępny jest obraz maszyny wirtualnej przeznaczonej dla VirtualBoxa, który zawiera instalację systemu Ubuntu Desktop ze skonfigurowanym serwerem Algocheckera, na której z łatwością można uruchomić przykłady prezentowane w artykule i z nimi poeksperymentować.

PODSUMOWANIE

W artykule opisaliśmy budowę skalowalnej platformy do testowania algorytmów zbudowanej w oparciu o Dockera. Projekt Algochecker jest obecnie testowany na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Znajduje zastosowanie do oceny kolokwium oraz zadań laboratoryjnych z programowania w C i C++.

Kod źródłowy projektu oraz demo dostępne są na stronie poświęconej projektowi [11]. Projekt nadal jest dynamicznie rozwijany, choć funkcje wykorzystane w przykładach zawartych w tym artykule są już stabilne i gotowe do zastosowań praktycznych. Zapraszamy do zapoznania się z Algocheckerem i współpracy.

Michał Leszczyński, Dmytro Ievseienko, Paweł Paczuski, Przemysław Miazga

Przypisy:

[1] http://spoj.org/
[2] https://www.hackerrank.com/
[3] https://uva.onlinejudge.org/
[4] https://codility.com/
[5] https://redis.io/commands/blpop
[6] https://redis.io/topics/pubsub
[7] https://docs.docker.com/engine/reference/commandline/stats/
[8] https://en.wikipedia.org/wiki/Spinlock
[9] http://www.tldp.org/LDP/lpg/node15.html
[10] https://docs.docker.com/engine/security/security/
[11] https://algochecker.com/