P O L I T E C H N I K A Ś W I Ę T O K R Z Y S K A

Wydział Elektrotechniki, Automatyki i Informatyki

Maksymilian Sowula, Paweł Marek, Mateusz Nowak

Numer grupy dziekańskiej: 1ID21A

Łamacz haseł

Programowanie systemów rozproszonych - Projekt

1. Wstęp teoretyczny

Łamacz haseł – narzędzie służące do odgadnięcia hasła przypisanego do konkretnego konta lub konkretnych kont użytkowników systemu informatycznego celem osiągnięcia dostępu do chronionych danych.

1.1. Algorytmy powiązane z projektem

1.1.1. BruteForce

Jest to sposób łamania haseł polegający na dostarczaniu do systemu wszystkich możliwych kombinacji znaków do momentu znalezienia poprawnego hasła. Jest to czasochłonny proces, ponieważ jest to metoda posiadająca dużą złożoność obliczeniową przez co łamanie przez nią długich oraz skomplikowanych haseł jest prawie niemożliwe. Złożoność tego algorytmu można zapisać jako:

$$O(C^L)$$

gdzie:

- C – liczba możliwych znaków w haśle,
- L – długość hasła.

Przykładowo dla hasła złożonego z 4 znaków i z 10 cyfr (0-9) złożoność będzie wynosiła:

$O(C^L) = O(10^4) = O(10\ 000)$

Co oznacza, że aby złamać konkretne hasło złożone z kombinacji 4 znaków z 10 cyfr algorytm ten będzie musiał wykonać 10 000 prób.

1.1.2. Metoda Słownikowa

Jest to sposób łamania haseł polegający na próbie odgadnięcia hasła za pomocą zdefiniowanego słownika z najczęściej używanymi frazami, które użytkownicy wykorzystują jako hasła w systemach informatycznych. Nie testuje się tutaj wszystkich kombinacji haseł tak jak w Brute Force, jednakże sprawdza się tylko popularne hasła. Złożoność obliczeniową tego algorytmu można zapisać jako:

$$O(S)$$

gdzie:

- S – ilość słów w słowniku.

W przypadku próby złamania hasła z użyciem słownika zawierającego 10 000 słów złożoność będzie wynosiła:

$$O(10\ 000)$$

Co oznacza, że próba złamania hasła zostanie wykonana 10 000 razy czyli dokładnie tyle, ile słów znajduje się w słowniku.

1.1.3. MD5

Jest to algorytm kryptograficzny generującym 128 bitowy skrót z dowolnej długości wejściowego ciągu znaków. Algorytm ten tworzy hash słowa w postaci ciągu szesnastkowego. Algorytm ten nie jest odporny na ataki typu bruteforce oraz na zjawisko kolizji czyli możliwości utworzenia tego samego hashu dla dwóch różnych słów. Jego złożoność obliczeniowa to:

$$O(n)$$

gdzie:

- n – liczba bitów w danych wejściowych.

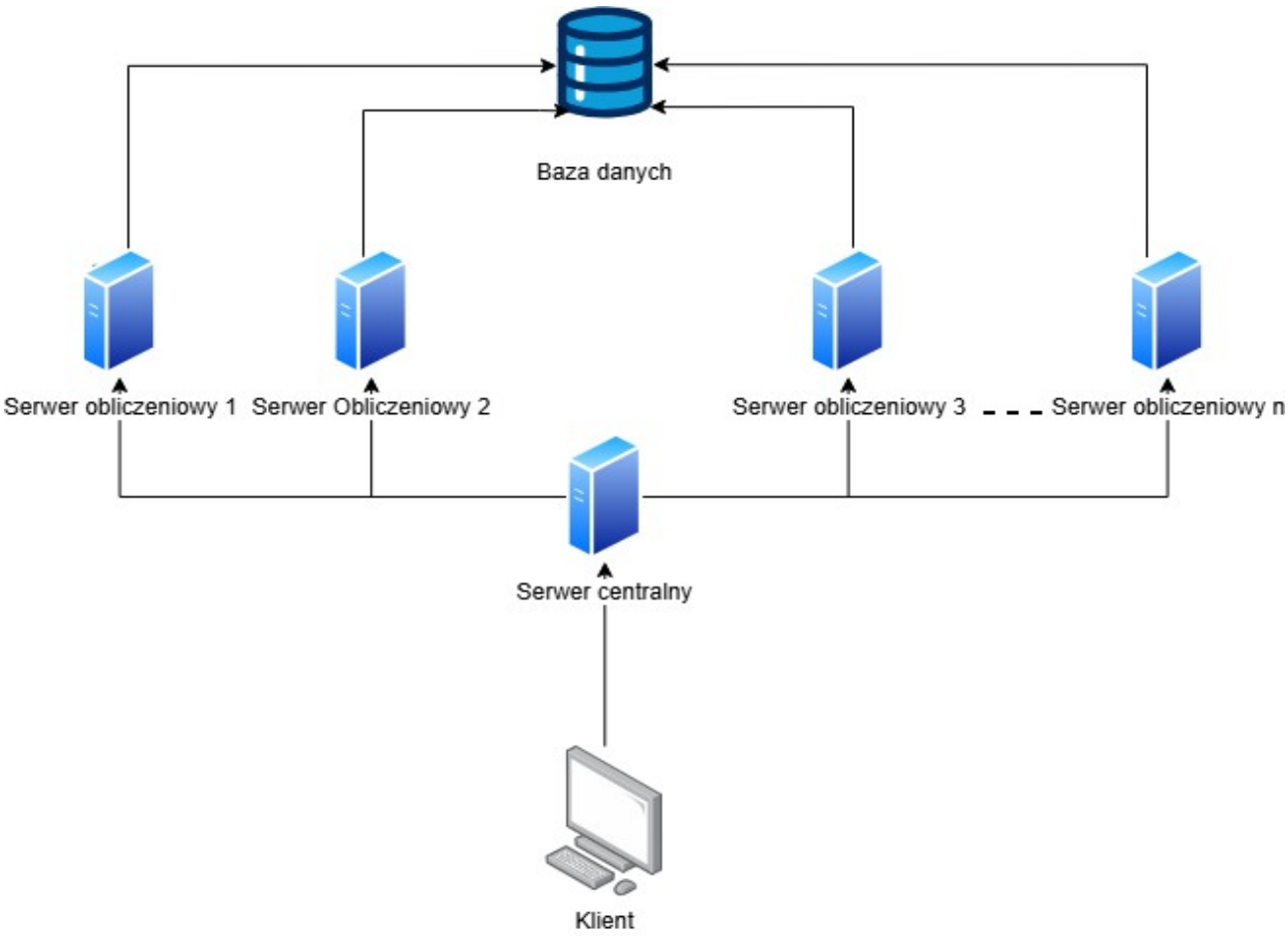
W przypadku 512 bitów zawartych w danych wejściowych złożoność tego algorytmu będzie wynosiła:

$$O(512)$$

Co oznacza, że blok danych o długości 512 bitów zostanie wykonany w stałym czasie ponieważ będzie wymagał stałej liczby operacji w kontekście hashującego algorytmu MD5.

1.2. Architektura rozwiązania

Główna architektura rozwiązania będzie opierała się na architekturze Klient ↔ Serwer. Dokładniej precyzując klient będzie komunikował się z serwerem centralnym, który będzie rozpraszał obliczenia po innych hostach oczekując na rezultat. Liczba serwerów (n) ustalana będzie na podstawie wartości przekazanej od klienta. Rysunek 1.1 przedstawia uproszczoną architekturę aplikacji.



Rysunek 1.1. Uproszczona architektura aplikacji.

1.2.1 Baza danych

Dla celi projektowych utworzono bazę danych, w której zawarto wyłącznie 1 encję – „*Users*” (rys 1.2) zawierającą pola:

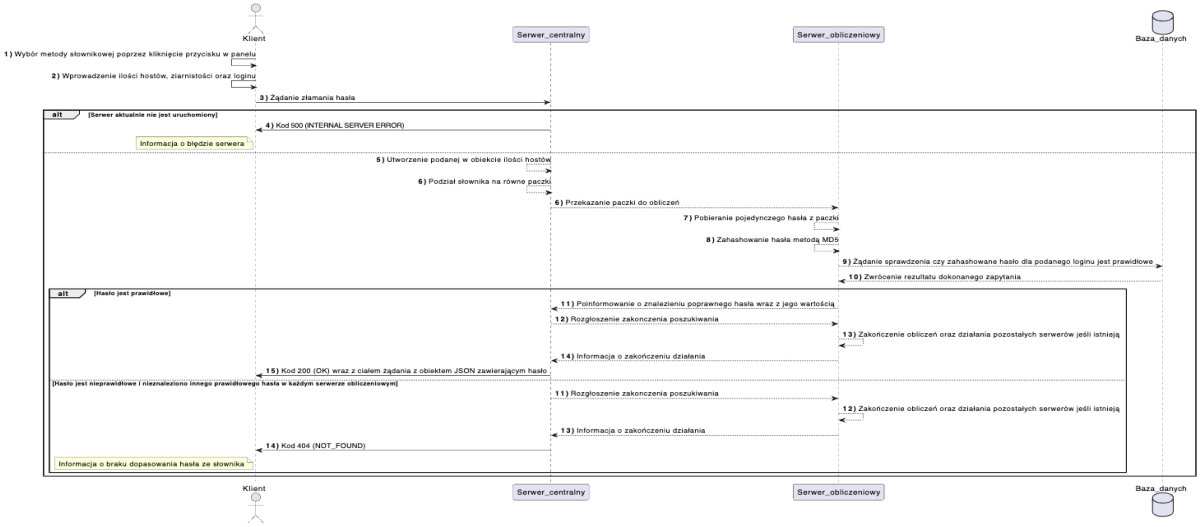
- *id* – identyfikator konta użytkownika,
- *login* – login użytkownika,
- *password* – hash uzyskany metodą MD5 hasła użytkownika.

Users	
PK	id: NUMBER
	login: VARCHAR
	password: VARCHAR

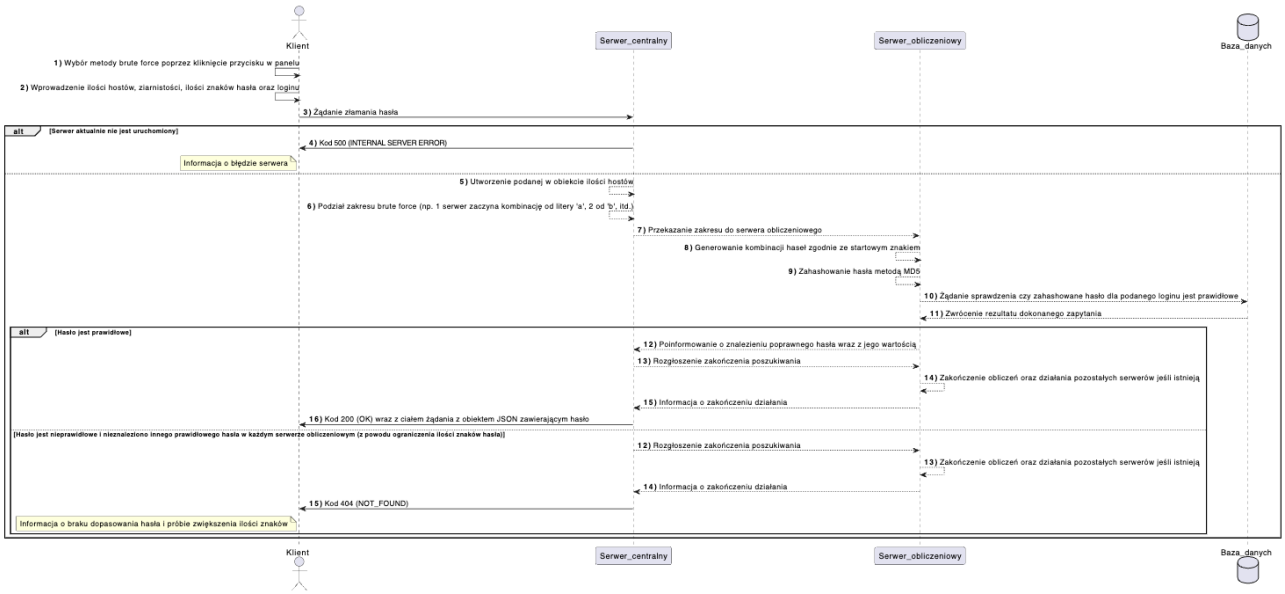
Rysunek 1.2. Encja *Users*.

1.2.2. Przypadki użycia

W tworzonej aplikacji przewidziano dwa przypadki użycia przypisane do dwóch metod łamania haseł – słownikowej oraz brute force. Na rysunkach poniżej przedstawiono przebieg komunikacji między klientem, serwerami oraz bazą danych.



Rysunek 1.3. Przypadek użycia łamania hasła metodą słownikową.



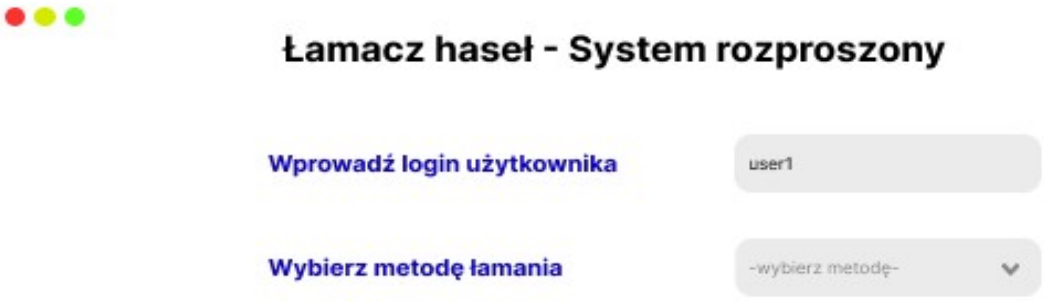
Rysunek 1.4. Przypadek użycia łamania hasła metodą brute force.

1.2.3. Projekt interfejsu graficznego

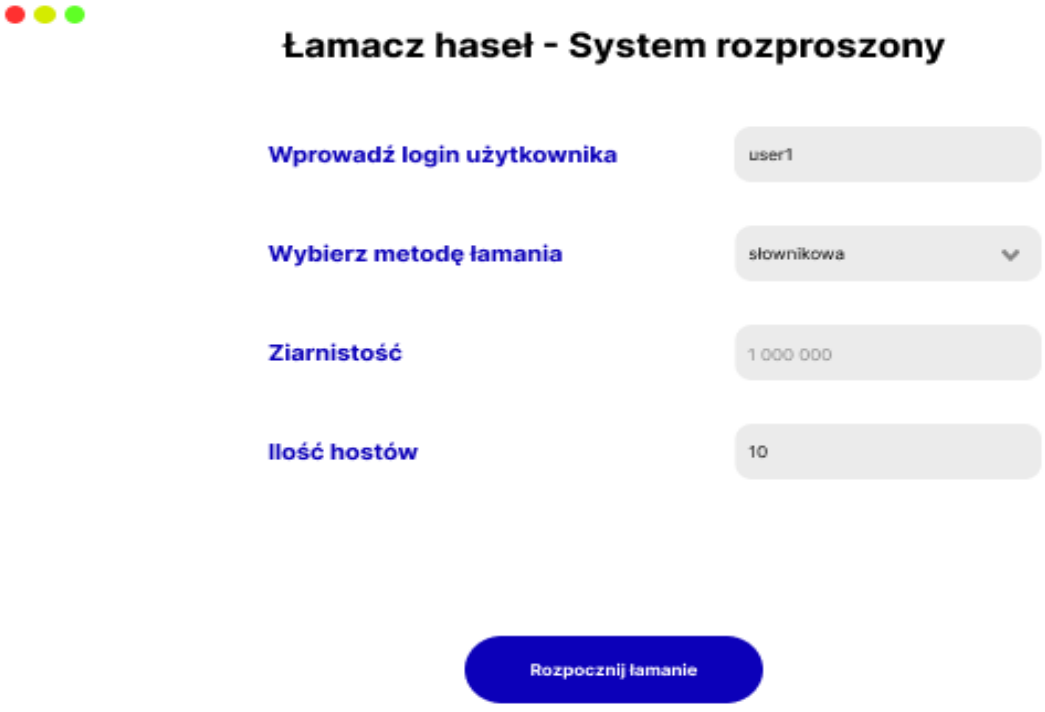
W projekcie interfejsu graficznego aplikacji przewidziano 3 ekrany:

- Ekran wprowadzania loginu użytkownika, którego hasło chcemy złamać wraz z wyborem metody,
- Ekran wprowadzania liczby hostów oraz automatycznego ustawiania ziarnistości w zależności od liczby hostów i wielkości paczki (input z właściwością read only) oraz w przypadku ataku metodą słownikową wielkości paczki lub w przypadku ataku metodą brute force limit ilości znaków po którym algorytm ma zatrzymać swoje działanie jeśli wykorzystane zostaną wszystkie kombinacje,
- Ekran przedstawienia rezultatu łamania hasła. W przypadku pozytywnego rezultatu wyświetlony zostanie hash hasła wraz z jego wartością i statystykami w postaci czasu łamania hasła wraz z liczbami prób. W przypadku niepowodzenia zostanie wyświetlony komunikat o niepowodzeniu wraz z informacją o użytej metodzie i ze statystykami, które zostały wymienione przy ekranie z pozytywnymi rezultatami.

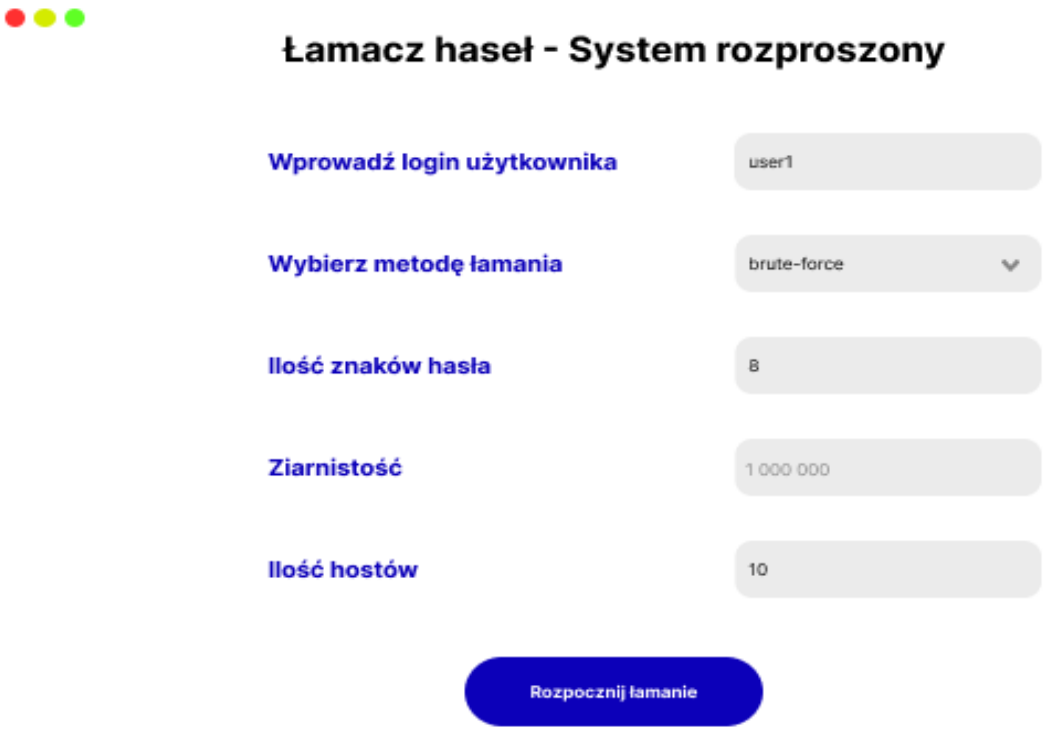
Poniższe rysunki przedstawiają projekt graficzny wymienionych ekranów aplikacji:



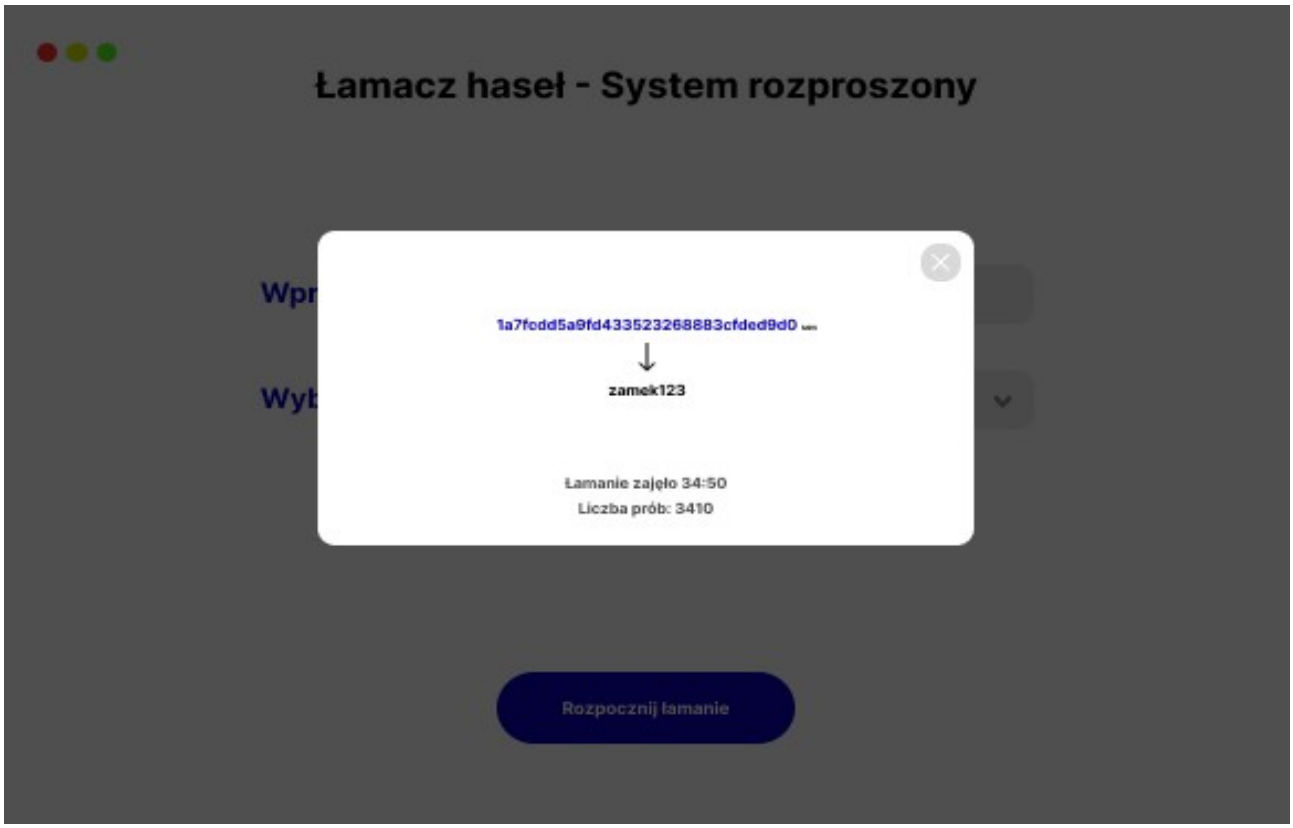
Rysunek 1.5. Ekran wyboru metody i wprowadzania loginu użytkownika.



Rysunek 1.6. Ekran formularza szczegółów dotyczących rozproszenia obliczeń w metodzie słownikowej.



Rysunek 1.7. Ekran formularza szczegółów dotyczących rozproszenia obliczeń w metodzie brute force.



Rysunek 1.7. Ekran przedstawiający uzyskany rezultat.

1.3. Sposób testowania

Aplikacja zostanie wytestowana poprzez sprawdzenie wpływu ustawienia parametrów w postaci liczby hostów oraz ziarnistości na efektywność oraz przyspieszenie dla obu metod łamania haseł.

1.3.1. Testy przyspieszenia

Przyspieszenie to stosunek czasu wykonania obliczeń na jednym hoście do czasu wykonania obliczeń na N hostach. Wzór stosowany do wyznaczania przyspieszania będzie następujący:

$$S(N) = \frac{T(1)}{T(N)}$$

gdzie:

- S – przyspieszenie (ang. speedup),
- N – ilość hostów,
- T – czas wykonania obliczeń.

1.3.2. Testy efektywności

Efektywność to stosunek wyznaczonego wcześniej przyspieszenia do liczby użytych hostów wyrażający wydolność wykorzystania dostępnych zasobów obliczeniowych. Do jej wyznaczania zostanie użyty następujący wzór:

$$E(N) = \frac{S(N)}{N} = \frac{T(1)}{N * T(N)}$$

gdzie:

- E – efektywność (ang. speedup),
- S – przyspieszenie (ang. efficiency),
- N – ilość hostów,
- T – czas wykonania obliczeń.

Wyniki testów zostaną zaprezentowane w postaci tabelki wizualizującej stosunek czasu wykonania obliczeń w zależności od ustawionej ziarnistości oraz użytej liczby hostów oraz tabelki wizualizującej wpływ liczby hostów na przyspieszenie i efektywność. Przykładowe tabelki przedstawiono poniżej:

Tabela 1.1. Przykładowa tabela przedstawiająca porównanie wpływu ziarnistości i liczby procesorów na czas wykonanych obliczeń.

Liczba procesorów (N)	**Ziarnistość Z = 1**	**Ziarnistość Z = 10**	**Ziarnistość Z = 100**
1	100 sekund	10 sekund	1 sekunda
4	25 sekund	2,5 sekundy	0,25 sekundy

Tabela 1.2. Przykładowa tabela przedstawiająca porównanie wpływu liczby procesorów na przyspieszenie i efektywność.

Liczba procesorów (N)	**Przyspieszenie**	**Efektywność**
1	S(1) = 1	E(1) = 1
4	S(4) = 4	E(4) = 1

2. Implementacja

2.1 Frontend

2.1.1 Definicja schematu formularza

W celu zapewnienia poprawności wprowadzanych danych, formularz został poddany walidacji z wykorzystaniem biblioteki Zod. Poniższy fragment kodu definiuje strukturę danych niezbędnych do przeprowadzenia procesu łamania haseł. Schemat zawiera nazwę użytkownika, metodę ataku (brute-force lub słownikową), opcjonalną długość hasła oraz plik słownikowy, jeśli metoda tego wymaga.

```
const formSchema = z.object({
  username: z
    .string()
    .min(2, { message: "Nazwa użytkownika musi mieć co najmniej 2 znaki." }),
  method: z.string(),
  passwordLength: z.coerce
    .number()
    .gte(2, "Hasło musi mieć co najmniej 2 znaków.")
    .optional(),
  dictionaryFile: z
    .instanceof(File)
    .refine((file) => file.name.endsWith(".txt"), {
      message: "Plik musi być w formacie .txt.",
    })
    .optional(),
});
```

Listing 2.1. Fragment kodu przedstawiający definicję schematu formularza z walidacją danych wejściowych przy użyciu biblioteki Zod.

2.1.2 Obsługa przesyłania formularza

Proces obsługi formularza zależy od wybranej przez użytkownika metody ataku: brute force lub ataku słownikowego. Dane przesyłane są w formacie JSON lub jako multipart/form-data, w zależności od kontekstu.

```
const onSubmit: SubmitHandler<FormData> = async (data) => {
```

```
  setDialogOpen(true);
  setIsLoading(true);

  try {
    let response: AxiosResponse<CrackingResponse> | undefined;
    if (data.method === "brute-force") {
      response = await axios.post(`${BASE_URL}/cracking/brute-force`, {
        userLogin: data.username,
        passwordLength: data.passwordLength,
      });
    } else if (data.method === "słownikowa") {
      const formData = new FormData();
      formData.append("username", data.username);
      if (data.dictionaryFile) {
        formData.append("file", data.dictionaryFile);
        await axios.post(`${BASE_URL}/synchronizing/dictionary`, formData, {
          headers: { "Content-Type": "multipart/form-data" },
        });
      }
      response = await axios.post(
        `${BASE_URL}/cracking/dictionary`,
        formData,
        { headers: { "Content-Type": "multipart/form-data" } }
      );
    }
    setResponse(response?.data);
  } catch (error) {
    // obsługa błędów
  } finally {
    setIsLoading(false);
  }
};
```

Listing 2.2. Fragment kodu przedstawiający obsługę przeciągania pliku i jego walidację przed przesłaniem na serwer.

2.1.3 Renderowanie interfejsu formularza

Interfejs użytkownika zawiera intuicyjny formularz umożliwiający wybór metody ataku oraz wprowadzenie odpowiednich parametrów. Formularz jest responsywny i posiada nowoczesny design.

```
return (
  <>
```

```
    <form
      onSubmit={handleSubmit(onSubmit, (errors) => {
        // obsługa błędów
      })}
      className="w-full max-w-xl mx-auto min-h-[80vh] flex flex-col justify-between"
    >
      <div className="space-y-6">
        <div className="text-center">
          <h1 className="text-xl md:text-4xl font-bold mb-24 ">
            Łamacz haseł - System rozproszony
          </h1>
        </div>

        {/* Pola formularza */}
      </div>

      <div className="pt-4 flex justify-center">
        {methodSelected && (
          <Button
            type="submit"
            className="bg-primary rounded-full p-6 cursor-pointer"
          >
            Rozpocznij łamanie
          </Button>
        )}
      </div>
    </form>

    <PasswordCrackerDialog
      open={dialogOpen}
      setOpen={setDialogOpen}
      response={response}
      isLoading={isLoading}
    />
  </>
);
```

Listing 2.3. Fragment kodu przedstawiający strukturę i renderowanie formularza interfejsu użytkownika.

2.1.4 Obsługa przeciągania i upuszczania plików

Obsługa funkcjonalności „drag and drop” umożliwia dodanie pliku słownikowego do formularza w sposób intuicyjny. Dodatkowo zaimplementowana została walidacja typu pliku oraz aktualizacja stanu komponentu po jego załadowaniu.

```
const handleDrop = (e: React.DragEvent<HTMLDivElement>) => {
  e.preventDefault();
  const file = e.dataTransfer.files?.[0];
  if (file) {
    if (!file.name.endsWith(".txt")) {
      toast("Plik musi być w formacie .txt.", {
        style: { background: "red", color: "white" },
      });
      return;
    }
    if (fileInputRef.current) {
      const dataTransfer = new DataTransfer();
      dataTransfer.items.add(file);
      fileInputRef.current.files = dataTransfer.files;
    }
    setSelectedFileName(file.name);
  }
};
```

Listing 2.4. Fragment kodu przedstawiający obsługę przeciągania i upuszczania plików słownikowych do formularza.

2.1.5 Wyświetlanie wyników łamania hasła

Po zakończeniu procesu łamania użytkownik otrzymuje szczegółowe informacje dotyczące jego przebiegu. Dane te wyświetlane są w formie kart, przedstawiających między innymi całkowity czas wykonania operacji, średni czas odpowiedzi serwerów oraz czas komunikacji.

```
<Card className="relative overflow-hidden">
  <ShineBorder shineColor={["#A07CFE", "#FE8FB5", "#FFBE7B"]} />
  <CardHeader>
    <CardTitle>Czas wykonania</CardTitle>
  </CardHeader>
  <CardContent>
    <p>Całkowity czas: {response.totalExecutionTime} ms</p>
    <p>Średni czas serwera: {response.averageServerTime} ms</p>
    <p>Czas komunikacji: {response.communicationTime} ms</p>
  </CardContent>
</Card>
```

Listing 2.5. Fragment kodu przedstawiający komponent odpowiedzialny za prezentację metryk czasowych procesu łamania hasła.

2.1.6 Wizualizacja czasów poszczególnych serwerów

Dodatkowo aplikacja umożliwia analizę efektywności poszczególnych węzłów w systemie rozproszonym. Prezentowane są szczegółowe dane czasowe dotyczące każdego z serwerów biorących udział w operacji.

```
<Card className="relative overflow-hidden">
  <ShineBorder shineColor={["#A07CFE", "#FE8FB5", "#FFBE7B"]} />
  <CardHeader>
    <CardTitle>Czasy serwerów</CardTitle>
  </CardHeader>
  <CardContent>
    <ul className="list-disc pl-5">
      {response.serversTimes.map((serverTime, index) => (
        <li key={index}>
          Serwer {serverTime.server}: {serverTime.time} ms
        </li>
      ))}
    </ul>
  </CardContent>
</Card>
```

Listing 2.6. Fragment kodu przedstawiający wizualizację czasów odpowiedzi serwerów uczestniczących w procesie łamania hasła.

2.2 Serwer centralny

2.2.1 Architektura systemu i koordynacja serwerów

Zadania w systemie rozproszonym koordynowane są przez klasę TaskCoordinatorService, odpowiedzialną za przydzielanie zadań serwerom obliczeniowym, monitorowanie ich stanu oraz anulowanie zadań w przypadku odnalezienia hasła.

```
public class TaskCoordinatorService
{
    public static PasswordInfo? LastFoundPassword { get; private set; }
    private volatile bool passwordFound;
    private HashSet<IPAddress> failedServers;
    private readonly IEnumerable<ILogService> logServices;
    private Dictionary<CalculatingServerState, Task> processingTasks;
```

```
    private Dictionary<CalculatingServerState, TaskCompletionSource<bool>> taskCompletionSources;

    public void AddServerTask(CalculatingServerState server, Chunk chunk, string username)
    {
        TaskCompletionSource<bool> taskCompletionSource = new();
        taskCompletionSources[server] = taskCompletionSource;
           processingTasks[server] = ProcessServerChunkAsync(server, chunk, taskCompletionSource,
username);
    }

    public void CancelAllTasks()
    {
        foreach (TaskCompletionSource<bool> taskCompletionSource in taskCompletionSources.Values)
        {
            taskCompletionSource.TrySetCanceled();
        }
    }
}
```

Listing 2.7. Fragment kodu przedstawiający klasę koordynującą serwery i zadania.

2.2.2 Konfiguracja serwera centralnego

Konfiguracja serwera centralnego obejmuje rejestrację niezbędnych usług, ustawienia dotyczące parametrów granulacji dla metod łamania haseł oraz konfigurację polityki CORS, umożliwiającej komunikację z frontendem oraz serwerami obliczeniowymi.

```
public class Startup
{
    private IEnumerable<ILogService>? logServices;
    public static bool IsDatabaseRunning { get; private set; } = false;
    public static List<IPAddress> ServersIpAddresses { get; set; } = new List<IPAddress>();
    public static int DictionaryGranularity { get; set; } = 10000;
    public static int BruteForceGranularity { get; set; } = 10;

    public void ConfigureServices(IServiceCollection services)
    {
        if (services != null)
        {
            services.AddControllers();
            services.AddScoped<DictionarySynchronizingService>();
            services.AddScoped<ILogService, InfoLogService>();
            services.AddScoped<ILogService, ErrorLogService>();
            services.AddScoped<ICrackingService, CrackingService>();
            services.AddScoped<IDictionarySynchronizingService, DictionarySynchronizingService>();
```

```
            services.AddScoped<ICheckService, CheckService>();
            services.AddScoped<CheckService>();
            services.AddScoped<IResponseProcessingService, ResponseProcessingService>();
            services.AddScoped<IServerCommunicationService, ServerCommunicationService>();
            services.AddScoped<IBruteForceCrackingService, BruteForceCrackingService>();
            services.AddScoped<IDictionaryCrackingService, DictionaryCrackingService>();
            services.AddScoped<ICrackingService, CrackingService>();
            // konfiguracja CORS
            ConfigureCors(services);
        }
    }

    private static void ConfigureCors(IServiceCollection services)
    {
        services.Configure<Microsoft.AspNetCore.Http.Features.FormOptions>(options =>
        {
            options.MultipartBodyLengthLimit = 32212254720;
        });
        services.AddCors(options =>
        {
            options.AddPolicy("AllowedOrigins", policy =>
                policy.SetIsOriginAllowed(origin =>
                {
                    Uri uri = new(origin);
                    return uri.Port == 5099 || uri.Port == 5173;
                })
                .AllowAnyHeader()
                .AllowAnyMethod()
            );
        });
    }
}
```

Listing 2.8. Fragment kodu przedstawiający konfigurację aplikacji.

2.2.3 Metoda brute force - dystrybucja znaków

Algorytm dystrybucji znaków w metodzie brute force dzieli dostępny zestaw znaków na mniejsze porcje zgodnie z określonym poziomem granulacji. Tak przygotowane porcje są następnie przekazywane do serwerów obliczeniowych, co umożliwia równoległe przetwarzanie danych.

```
public CrackingCharPackage DistributeCharacters()
{
    string allChars = GetAllAvailableCharacters();
    int totalCharCount = allChars.Length;
    int granularity = GetGranularity();
    int portionSize = CalculatePortionSize(totalCharCount, granularity);
```

```
    List<string> charPortions = CreateCharacterPortions(allChars, totalCharCount, portionSize);
    return new CrackingCharPackage(charPortions);
}

private static string GetAllAvailableCharacters()
{
    return "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
}

private static int GetGranularity()
{
    return Startup.BruteForceGranularity > 0 ? Startup.BruteForceGranularity : 5;
}

private static List<string> CreateCharacterPortions(string allChars, int totalCharCount, int
portionSize)
{
    List<string> charPortions = new();
    for (int i = 0; i < totalCharCount; i += portionSize)
    {
        int currentPortionSize = Math.Min(portionSize, totalCharCount - i);
        string portion = allChars.Substring(i, currentPortionSize);
        charPortions.Add(portion);
    }
    return charPortions;
}
```

Listing 2.9. Fragment kodu przedstawiający dystrybucję znaków w metodzie brute force.

2.2.4 Obsługa żądań łamania hasła metodą brute force

Żądania związane z łamaniem hasła metodą brute force obsługiwane są przez metodę, która odpowiada za analizę danych wejściowych, przygotowanie pakietów znaków, uruchomienie zadań na serwerach oraz przetworzenie uzyskanych wyników.

```
public async Task<IActionResult> HandleBruteForceRequest(HttpContext httpContext)
{
    DateTime startTime = DateTime.UtcNow;
    LogBruteForceStart();
    try
    {
        string bodyContent = await ReadRequestBody(httpContext);
        BruteForceRequestData credentials = ParseAndValidateRequest(bodyContent);
        CrackingCharPackage charPackage = DistributeCharacters();
```

```
        ServerTaskResult taskResult = await ExecuteTasks(charPackage, credentials);
        int totalTime = (int)(DateTime.UtcNow - startTime).TotalMilliseconds;
        CrackingResult? successResult = taskResult.Results.FirstOrDefault(r => r.Success);
        bool passwordFound = successResult != null && successResult.Success;
        string? serverIp = successResult?.ServerIp;
            return responseProcessingService.ProcessResults(taskResult, credentials, charPackage,
totalTime);
    }
    catch (Exception ex)
    {
        int errorTime = (int)(DateTime.UtcNow - startTime).TotalMilliseconds;
        return responseProcessingService.HandleError(ex, errorTime);
    }
}
```

Listing 2.10. Fragment kodu przedstawiający obsługę żądań łamania hasła metodą brute force.

2.2.5 Wykonywanie zadań na serwerach obliczeniowych

Proces równoległego wykonywania zadań na wielu serwerach obejmuje przygotowanie zadań, oczekiwanie na ich wykonanie oraz pomiar czasów realizacji. Zgromadzone dane służą do analizy wydajności działania systemu.

```
public async Task<ServerTaskResult> ExecuteTasks(CrackingCharPackage charPackage,
BruteForceRequestData credentials)
{
    DateTime tasksStartTime = DateTime.UtcNow;
    List<string> serverIPs = GetServerIPs();
    ValidateServerCount(serverIPs);
    List<Task<CrackingResult>> tasks = await serverCommunicationService.CreateTasksForPortions(
        charPackage.CharPortions,
        credentials.PasswordLength,
        credentials.UserLogin,
        serverIPs);
    DateTime awaitStartTime = DateTime.UtcNow;
    int taskSetupTime = (int)(awaitStartTime - tasksStartTime).TotalMilliseconds;
    CrackingResult[] results = await ExecuteAllTasks(tasks);
    DateTime resultsTime = DateTime.UtcNow;
    int processingTime = (int)(resultsTime - awaitStartTime).TotalMilliseconds;
    LogTaskProcessingCompleted(results.Length, serverIPs.Count, processingTime);
    return new ServerTaskResult(taskSetupTime, processingTime, results);
}
```

Listing 2.11. Fragment kodu przedstawiający wykonywanie zadań na serwerach obliczeniowych.

2.2.6 Przetwarzanie wyników łamania hasła

Wyniki zwracane przez serwery są analizowane w celu określenia, czy któremuś z nich udało się odnaleźć hasło. W przypadku powodzenia generowana jest odpowiednia odpowiedź, natomiast w przypadku niepowodzenia zwracana jest informacja o braku wyniku.

```
public IActionResult ProcessResults(ServerTaskResult taskResult, BruteForceRequestData
credentials, CrackingCharPackage charPackage, int totalTime)
{
    CrackingResult? successfulResult = FindSuccessfulResult(taskResult.Results);
    if (successfulResult != null && successfulResult.Success)
    {
            return CreateSuccessResponse(taskResult, successfulResult, credentials, charPackage,
totalTime);
    }
    return CreateNotFoundResponse(taskResult, credentials, charPackage, totalTime);
}

private static CrackingResult? FindSuccessfulResult(IEnumerable<CrackingResult> results)
{
    return results.FirstOrDefault(static result => result.Success);
}

private IActionResult CreateSuccessResponse(ServerTaskResult taskResult, CrackingResult
successfulResult, BruteForceRequestData credentials, CrackingCharPackage charPackage, int
totalTime)
{
    int communicationTime = totalTime - successfulResult.Time;
            LogSuccessfulCentralTiming(totalTime, successfulResult.Time, communicationTime,
successfulResult.ServerIp);
    return new OkObjectResult(CreateSuccessResponseObject(
        successfulResult, totalTime, communicationTime,
        credentials, charPackage, taskResult));
}
```

Listing 2.12. Fragment kodu przedstawiający przetwarzanie wyników łamania hasła.

2.2.7 Rejestrowanie metryk wydajnościowych

Metryki związane z procesem łamania haseł zapisywane są w pliku CSV. Rejestrowane dane obejmują informacje dotyczące użytkownika, długości hasła, wykorzystanych znaków, adresu serwera, czasów przetwarzania oraz powodzenia operacji.

```
public static void LogBruteForcePackageMetrics(
    IEnumerable<ILogService> logServices,
    string userLogin,
    int passwordLength,
    string charPackage,
    string serverIp,
    int processingTime,
    int totalTime,
    bool passwordFound,
    int granularity)
{
    try
    {
        // zapis metryki do pliku CSV
        lock (BruteForceFileLock)
        {
            string path = BruteForceMetricsPath;
            bool fileExists = File.Exists(path);
            using StreamWriter writer = new(path, true);
            if (!fileExists)
            {
                writer.WriteLine("Login,PasswordLength,CharPackage,ServerIp,ProcessingTime,TotalTime,PasswordFound,Granularity");
            }
            writer.WriteLine($"{userLogin}," +
                        $"{passwordLength}," +
                        $"{charPackage}," +
                        $"{serverIp}," +
                        $"{processingTime}," +
                        $"{totalTime}," +
                        $"{passwordFound}," +
                        $"{granularity}");
        }

        ILogService.LogInfo(logServices, $"Brute force package metrics saved to {BruteForceMetricsPath}");
    }
    catch (Exception ex)
    {
        ILogService.LogError(logServices, $"Failed to log brute force package metrics: {ex.Message}");
    }
}
```

Listing 2.13. Fragment kodu przedstawiający rejestrowanie metryk wydajnościowych.

2.2.8 Obsługa awarii serwerów obliczeniowych

System identyfikuje serwery, które uległy awarii, i usuwa je z dalszego przetwarzania. Serwery oznaczone jako uszkodzone nie są uwzględniane przy kolejnych próbach, co zwiększa odporność systemu na błędy i awarie.

```
private void HandleProcessingException(CalculatingServerState server, Exception ex,
TaskCompletionSource<bool> taskCompletionSource)
{
    if (ex is PasswordFoundException)
    {
        passwordFound = true;
        CancelAllTasks();
        taskCompletionSource.TrySetException(ex);
        return;
    }
    if (!taskCompletionSource.Task.IsCanceled && !passwordFound)
    {
        HandleServerError(server);
        taskCompletionSource.TrySetException(ex);
    }
}

private void HandleServerError(CalculatingServerState server)
{
    if (!failedServers.Contains(server.IpAddress))
    {
        failedServers.Add(server.IpAddress);
        ILogService.LogInfo(logServices, $"Server {server.IpAddress} marked as failed");
    }
}

private void MarkServerAsFailed(IPAddress serverIp)
{
    if (Startup.ServersIpAddresses.Contains(serverIp))
    {
        Startup.ServersIpAddresses.Remove(serverIp);
        failedServers.Add(serverIp);
        ILogService.LogInfo(logServices,
            $"Removed failed server {serverIp}. Remaining: {Startup.ServersIpAddresses.Count}");
    }
}
```

Listing 2.14. Fragment kodu przedstawiający obsługę awarii serwerów obliczeniowych.

2.2.9 Komunikacja z serwerami obliczeniowymi

Komunikacja pomiędzy serwerem centralnym a serwerami obliczeniowymi odbywa się za pomocą wysyłania żądań HTTP. Weryfikowana jest również dostępność poszczególnych serwerów, co pozwala zapewnić ciągłość działania systemu.

```
public async Task<HttpResponseMessage> SendRequestToServer(HttpClient httpClient, string serverIpAddress, string payloadJson)
{
    string url = $"http://{serverIpAddress}:5099/api/synchronizing/brute-force";
    StringContent content = new(payloadJson, Encoding.UTF8, "application/json");
    LogSendingBruteForceRequest(serverIpAddress);
    HttpResponseMessage response = await httpClient.PostAsync(url, content);
    return response;
}

public async Task ValidateServerConnection(string serverIpAddress)
{
    using HttpClient client = CreateHttpClient();
    string url = $"http://{serverIpAddress}:5099/api/calculating/check-connection";
    try
    {
        HttpResponseMessage response = await client.GetAsync(url);
        if (!response.IsSuccessStatusCode)
        {
                    throw new Exception($"Server {serverIpAddress} responded with status code {response.StatusCode}");
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"Failed to connect to server {serverIpAddress}: {ex.Message}");
    }
}
```

Listing 2.15. Fragment kodu przedstawiający komunikację z serwerami obliczeniowymi.

2.2.10 Formatowanie odpowiedzi do klienta

Odpowiedzi generowane przez system zawierają szczegółowe informacje o wynikach łamania hasła, w tym czasy wykonania poszczególnych etapów. Dane te są następnie prezentowane użytkownikowi w interfejsie graficznym aplikacji.

```
private static object CreateSuccessResponseObject(CrackingResult successfulResult, int totalTime, int communicationTime, BruteForceRequestData credentials, CrackingCharPackage charPackage, ServerTaskResult taskResult)
{
    return new
    {
        Message = "Password found.",
        successfulResult.Password,
        Server = successfulResult.ServerIp,
        ServerExecutionTime = successfulResult.Time,
        TotalExecutionTime = totalTime,
        CommunicationTime = communicationTime,
        Timing = new
        {
            credentials.ParseTime,
            taskResult.TaskSetupTime,
            taskResult.ProcessingTime,
            TotalTime = totalTime
        }
    };
}

private static object CreateNotFoundResponseObject(int totalTime, int avgServerTime, int avgCommunicationTime, List<CrackingResult> validResults, BruteForceRequestData credentials, CrackingCharPackage charPackage, ServerTaskResult taskResult)
{
    return new
    {
        Message = "Password not found by any server.",
        TotalExecutionTime = totalTime,
        AverageServerTime = avgServerTime,
        CommunicationTime = avgCommunicationTime,
        ServersTimes = validResults
            .Select(result => new { Server = result.ServerIp, result.Time })
            .ToList(),
        Timing = new
        {
            credentials.ParseTime,
            taskResult.TaskSetupTime,
            taskResult.ProcessingTime,
            TotalTime = totalTime
        }
    };
}
```

Listing 2.16. Fragment kodu przedstawiający formatowanie odpowiedzi do klienta.

2.3 Serwer obliczeniowy

Serwer obliczeniowy jest kluczowym elementem systemu rozproszonego do łamania haseł, odpowiedzialnym za wykonywanie operacji obliczeniowych związanych z metodami brute force oraz słownikową. Każdy serwer obliczeniowy komunikuje się z serwerem centralnym za pomocą protokołu HTTP, odbierając zadania w postaci fragmentów słownika lub zestawów znaków do przetworzenia, a następnie zwraca wyniki w formacie JSON. Poniżej przedstawiono szczegóły implementacji serwera obliczeniowego.

2.3.1 Konfiguracja serwera obliczeniowego

Konfiguracja serwera obliczeniowego została zaimplementowana w klasie CalculatingStartup.cs, która odpowiada za inicjalizację aplikacji oraz rejestrację usług. Serwer korzysta z platformy ASP.NET Core i jest skonfigurowany do nasłuchiwania na porcie 5099. Dodatkowo, ustawiono politykę CORS, umożliwiającą komunikację wyłącznie z serwerem centralnym działającym na porcie 5098.

```
public class CalculatingStartup

{

    public CalculatingStartup(IConfiguration configuration)

    {

        Env.Load("../.env");

        Configuration = configuration;

    }


    public void ConfigureServices(IServiceCollection services)

    {

        if (services != null)

        {

            ConfigureCors(services);

            services.AddSingleton<ILogService, InfoLogService>();
```

```
          services.AddSingleton<ILogService, ErrorLogService>();
            services.AddSingleton<IPasswordRepository, FilePasswordRepository>();

            services.AddScoped<CheckService>();

            services.AddScoped<DictionaryService>();

            services.AddScoped<ICheckService, CheckService>();

            services.AddScoped<IBruteForceService, BruteForceService>();

            services.AddScoped<IDictionaryService, DictionaryService>();

            services.AddControllers();

        }

    }


    private static void ConfigureCors(IServiceCollection services)

    {

        services.Configure<Microsoft.AspNetCore.Http.Features.FormOptions>(options =>

        {

            options.MultipartBodyLengthLimit = 32212254720;

        });

        services.AddCors(options =>

        {

            options.AddPolicy("AllowedOrigins", policy =>

                policy.SetIsOriginAllowed(origin =>

                {

                    Uri uri = new(origin);

                    return uri.Port == 5098;

                })

                .AllowAnyHeader()

                .AllowAnyMethod()

            );

        });

    }

}
```

Listing 2.17. Fragment kodu przedstawiający konfigurację serwera obliczeniowego.

2.3.2 Połączenie z serwerem centralnym

Serwer obliczeniowy nawiązuje połączenie z serwerem centralnym w celu rejestracji swojego adresu IP oraz potwierdzenia gotowości do wykonywania zadań. Proces ten jest realizowany w metodzie ConnectWithCentralServer w klasie CalculatingStartup.cs. Adresy IP serwera centralnego oraz serwera obliczeniowego są pobierane z pliku .env. W przypadku niepowodzenia połączenia serwer jest zatrzymywany, a odpowiedni komunikat zapisywany jest w logach.

```
private async Task ConnectWithCentralServer()
{
    try
    {
        Env.Load("../.env");
        string? centralServerIp = Env.GetString("CENTRAL_SERVER_IP");
        string? calculatingServerIp = Env.GetString("CALCULATING_SERVER_IP");
        if (string.IsNullOrEmpty(centralServerIp))
        {
            throw new Exception("CENTRAL_SERVER_IP is not set in the .env file");
        }
        if (string.IsNullOrEmpty(calculatingServerIp))
        {
            throw new Exception("CALCULATING_SERVER_IP is not set in the .env file");
        }
        using HttpClient httpClient = new();
        httpClient.Timeout = TimeSpan.FromSeconds(300);
        string centralServerUri = $"http://{centralServerIp}:5098/api/calculating-server/connect";
        using MultipartFormDataContent formData = new()
        {
            { new StringContent(calculatingServerIp), "IpAddress" }
        };
             HttpResponseMessage httpResponseMessage = await httpClient.PostAsync(centralServerUri,
formData);
```

```
        if (!httpResponseMessage.IsSuccessStatusCode)
        {
            throw new Exception($"Cannot connect to central server with address {centralServerIp}.
Status code: {httpResponseMessage.StatusCode}");
        }
        LogCalculatingServerInfo("Successfully connected to central server");
    }
    catch (Exception ex)
    {
        LogCalculatingServerError($"Error to connect to central server: {ex}");
        StopCalculatingServer();
    }
}
```

Listing 2.18. Fragment kodu przedstawiający proces nawiązywania połączenia z serwerem centralnym.

2.3.3 Łamanie haseł metodą słownikową

Metoda słownikowa jest implementowana w klasie DictionaryService.cs. Serwer obliczeniowy odbiera fragment słownika (określony zakres linii) od serwera centralnego, a następnie przetwarza go w celu odnalezienia hasła. Proces obejmuje wczytanie fragmentu słownika, obliczanie hashy MD5 dla każdego hasła oraz porównywanie ich z hashem docelowym przechowywanym w pliku users_passwords.txt. Wyniki są zwracane w formacie JSON.

```
public async Task<ActionResult> StartCrackingResult(HttpContext httpContext)
{
    DateTime startTime = DateTime.UtcNow;
    try
    {
        var (username, chunkInfo) = await ReadAndDeserializeRequest(httpContext);
        List<string> selectedPasswords = await ReadPasswordsFromDictionary(chunkInfo);
        ValidateSelectedPasswords(selectedPasswords, chunkInfo);
```

```
await CheckPasswordsAgainstDatabase(selectedPasswords, username);

        LogSuccessfulPasswordLoading(selectedPasswords, chunkInfo);

        DateTime endTime = DateTime.UtcNow;

        int processingTime = (int)(endTime - startTime).TotalMilliseconds;

        return JsonSuccessResponse("Password not found!", processingTime);

    }

    catch (Exception ex)

    {

        DateTime endTime = DateTime.UtcNow;

        int processingTime = (int)(endTime - startTime).TotalMilliseconds;

        if (ex.Message.Contains("Password found!"))

        {

            return JsonSuccessResponse(ex.Message, processingTime);

        }

        ILogService.LogError(logServices, $"Error while cracking using dictionary: {ex.Message}");

        return JsonErrorResponse(ex.Message, processingTime);

    }

}
```

Listing 2.19. Fragment kodu przedstawiający obsługę łamania hasła metodą słownikową.

2.3.4 Łamanie haseł metodą brute force

Metoda brute force jest implementowana w klasie BruteForceService.cs. Serwer obliczeniowy generuje wszystkie możliwe kombinacje znaków dla zadanego zestawu znaków i długości hasła, obliczając dla każdej kombinacji hash MD5 i porównując go z hashem docelowym. Proces jest ograniczony czasowo (60 sekund), aby zapobiec nadmiernemu zużyciu zasobów.

```
private string? PerformBruteForce(string chars, int passwordLength, string targetHash)

{

    DateTime bruteForceStartTime = DateTime.UtcNow;

       ILogService.LogInfo(logServices, $"Starting brute force with chars: '{chars}', length:
{passwordLength}, targetHash: {targetHash}");

    BigInteger combinationCount = 0;
```

```
  int charSetSize = chars.Length;
    BigInteger maxCombinations;

    if (passwordLength > 20)

    {

        maxCombinations = BigInteger.Pow(charSetSize, passwordLength);

    }

    else

    {

        maxCombinations = (long)Math.Pow(charSetSize, passwordLength);

    }

    const long MAX_COMBINATIONS_TO_CHECK = long.MaxValue;

    for (BigInteger i = 0; i < maxCombinations && i < MAX_COMBINATIONS_TO_CHECK; i++)

    {

        int elapsedTime = (int)(DateTime.UtcNow - bruteForceStartTime).TotalMilliseconds;

        if (elapsedTime >= TimeLimit)

        {

               ILogService.LogInfo(logServices, $"Brute force time limit reached ({TimeLimit}ms).
Stopping after {combinationCount} combinations");

            break;

        }

        combinationCount++;

        string combination = IndexToCombination(i, chars, passwordLength);

        string computedHash = CalculateMD5Hash(combination);

        if (computedHash == targetHash)

        {

            LogPasswordFound(combination, combinationCount, bruteForceStartTime);

            return combination;

        }

        LogProgressIfNeeded(combinationCount, bruteForceStartTime);

    }

    LogNoMatchFound(combinationCount, bruteForceStartTime);

    return null;

}
```

Listing 2.20. Fragment kodu przedstawiający implementację metody brute force.

2.3.5 Zarządzanie bazą haseł

Baza haseł jest przechowywana w pliku users_passwords.txt i zarządzana przez klasę FilePasswordRepository.cs. Klasa ta wczytuje hashe MD5 haseł użytkowników podczas inicjalizacji i umożliwia ich weryfikację w trakcie procesu łamania haseł.

```
public async Task Initialize()
{
    try
    {
                  ILogService.LogInfo(_logServices,  $"Initializing  password  repository  from:
{_passwordFilePath}");
        if (!File.Exists(_passwordFilePath))
        {
                           ILogService.LogError(_logServices,  $"Password  file  not  found  at:
{_passwordFilePath}");
            throw new FileNotFoundException($"Password file not found: {_passwordFilePath}");
        }
        FileInfo fileInfo = new (_passwordFilePath);
        ILogService.LogInfo(_logServices, $"Password file size: {fileInfo.Length} bytes");
        _userPasswords.Clear();
        int parsedCount = 0;
        using (StreamReader streamReader = new(_passwordFilePath))
        {
            string? line;
            while ((line = await streamReader.ReadLineAsync()) != null)
            {
                if (string.IsNullOrWhiteSpace(line))
                {
                    continue;
                }
                if (!line.Contains(':'))
                {
           ILogService.LogInfo(_logServices, $"Skipping invalid line (no colon): '{line}'");
```

```
   continue;
                }
                string[] parts = line.Split(':', 2);
                if (parts.Length == 2)
                {
                    string username = parts[0].Trim();
                    string hash = parts[1].Trim();
                    if (parsedCount < 10)
                    {
                           ILogService.LogInfo(_logServices, $"Adding user: '{username}' with hash:
'{hash}'");
                    }
                    _userPasswords[username] = hash;
                    parsedCount++;
                    if (parsedCount % 1000000 == 0)
                    {
                        ILogService.LogInfo(_logServices, $"Processed Kilkulet records so far...");
                    }
                }
            }
        }
        ILogService.LogInfo(_logServices, $"Loaded {parsedCount} valid user records from file");
    }
    catch (Exception ex)
    {
                  ILogService.LogError(_logServices,  $"Error  initializing  password  repository:
{ex.Message}");
        throw;
    }
}
```

Listing 2.21. Fragment kodu przedstawiający inicjalizację bazy haseł.

2.3.6 Rejestrowanie logów

Logi informacyjne i błędy są zapisywane za pomocą klas InfoLogService.cs oraz ErrorLogService.cs. Logi są zapisywane zarówno w konsoli, jak i w pliku, co ułatwia debugowanie i monitorowanie działania serwera.

```
public class InfoLogService : ILogService
{
    private string LogContent { get; set; } = "";

    public override void LogMessage(string message)
    {
        string timestamp = GetCurrentDate();
        LogContent = $"[INFO] {message} at [{timestamp}]";
        Console.WriteLine(LogContent);
        SaveToFile();
    }

    public override void SaveToFile()
    {
        try
        {
            File.AppendAllText(logFilePath, LogContent + Environment.NewLine);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ERROR] Failed to write [INFO] log to file: {ex.Message}");
        }
    }
}
```

Listing 2.22. Fragment kodu przedstawiający rejestrowanie logów informacyjnych.

2.3.7 Obsługa błędów

Serwer obliczeniowy został zaprojektowany z myślą o odporności na błędy. W przypadku problemów z połączeniem z bazą danych lub serwerem centralnym, serwer zatrzymuje swoje działanie, logując odpowiedni komunikat błędu. Mechanizm ten jest realizowany w metodzie StopCalculatingServer w klasie CalculatingStartup.cs.

```
private void StopCalculatingServer()
{
    if (app != null)
    {
        LogCalculatingServerInfo("Stopped calculating server");
                                                IHostApplicationLifetime      lifetime      =
app.ApplicationServices.GetRequiredService<IHostApplicationLifetime>();
        lifetime.StopApplication();
    }
}
```

Listing 2.23. Fragment kodu przedstawiający mechanizm zatrzymywania serwera w przypadku błędu.

3. Wyniki testów

Adres IP komputera	Wielkość przetwarzanej paczki	Typ łamania hasła	Czas wykonywania obliczeń (ms)	Całkowity czas wykonywania operacji (ms)	Granulacja	Numer próby

4. Wnioski