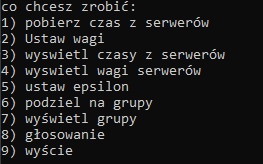
|  |
| --- |
| Politechnika Świętokrzyska w Kielcach  Wydział Elektrotechniki, Automatyki i Informatyki |
| Systemy odporne na błędy |
| Głosowanie przybliżone |
| Hubert Ptaszek, Mariusz Mularczyk |

1. **Cel pracy**

Celem pracy była symulacja pracy serwerów czasu połączonych w topologii gwiazdy z centralnym komputerem wyznaczającym w drodze głosowania przybliżonego najbardziej prawdopodobny czas. Program powinien dawać możliwość ustawienia wag przez użytkownika dla każdego komputera z osobna. Aplikacja została napisana z wykorzystaniem języka C#

1. **Opis aplikacji**

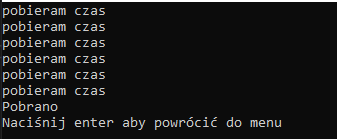
Po uruchomieniu aplikacji pojawi nam się menu z dostępnymi opcjami programu



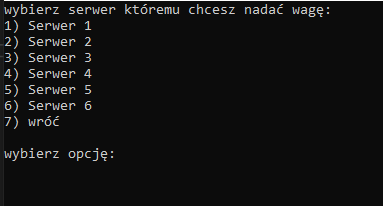
Po wybraniu pierwszej opcji system pobierze aktualny czas ze wszystkich serwerów. Operacja ta następuje w poniższy sposób:

|  |
| --- |
| public DateTime GetDateTime()          {              TimeSpan interval;              TimeSpan.TryParseExact("0.01", "s\\.ff", null, out interval);              Thread.Sleep(interval);              Console.WriteLine("pobieram czas");              return DateTime.Now;          } |

Dodatkowo jak możemy zauważyć usypiamy na chwilę watek przed wysłaniem aktualnego czasu aby wyniki czasowe były bardziej zróżnicowane. Po pobraniu czasów wyświetli nam się poniższy ekran



Program daje nam możliwość ustawienia wagi każdemu serwerowi z osobna



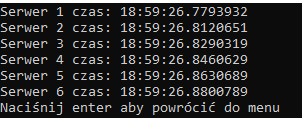
Waga serwera jest z zakresu od 1 do 10

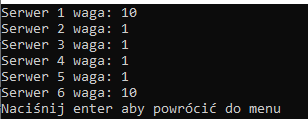


W przypadku wypisania liczby spoza zakresu program zwróci nam błąd i poprosi o ponowne wpisanie wagi



Program pozwala nam na wyświetlenie pobranych czasów oraz wag poszczególnych serwerów





Po wybraniu opcji nr 5 mamy możliwość ustawienia epsilony który w naszym przypadku odnosi się do milisekund.  
Wartość epsilonu powinna być dobrana odpowiednio do charakteru analizowanych danych: ponieważ:  
- zbyt małe ε może generować fałszywe alarmy;   
- zbyt duże ε może maskować niepoprawne dane.



Dzięki opcji nr 6 możemy pogrupować czasy serwerów zgodnie z podanym epsilonem. Odpowiada za to poniższa metoda.

|  |
| --- |
| public void  GroupTimes()          {              ServerList<Server> servers = new ServerList<Server>();              servers.Add(s1);              servers.Add(s2);              servers.Add(s3);              servers.Add(s4);              servers.Add(s5);              servers.Add(s6);              string[] formats = { "s\\.f", "s\\.ff", "s\\.fff", "s\\.ffff",                      "s\\.fffff", "s\\.ffffff", "s\\.fffffff", "s\\.ffffffff"};              TimeSpan interval;              TimeSpan.TryParseExact("0." + epsilon, formats, null, out interval);              groups = new Dictionary<int, ServerList<Server>>();              HashSet<ServerList<Server>> groupsLocal = new HashSet<ServerList<Server>>(new ServerListComparer());              foreach (Server server in servers) {                  TimeSpan time = server.Time.Value.TimeOfDay + interval;                  TimeSpan time2 = server.Time.Value.TimeOfDay - interval;                  List<Server> group = servers.Where(x => x.Time.Value.TimeOfDay >= time2 && x.Time.Value.TimeOfDay <= time).ToList();                  ServerList<Server> group2 = new ServerList<Server>(group);                  groupsLocal.Add(group2);              }              int iterator = 1;              foreach(ServerList<Server> item in groupsLocal)              {                  groups.Add(iterator, item);                  iterator++;              }          } |

Metoda działa w następujący sposób:   
Tworzymy listę serwerów a następnie dodajemy do niej nasze serwery następnie określamy formaty milisekund w dacie i staramy się sparsować nasz epsilon do typu TimeSpan dzięki któremu będziemy sprawdzać “odległości” pomiędzy naszymi czasami. Następnie deklarujemy samodzielnie zaimplementowany zbiór który pozwoli nam zachować unikalność grup. Następnie iterujemy po utworzonej wcześniej liście serwerów i sprawdzamy które serwey(a dokładnie ich czas) z naszej listy “leżą” w odległości *epsilon* od aktualnie sprawdzanego serwera. Serwery spełniające warunek dodawane są do zbioru jako grupa. Po zakończeniu iteracji przepisujemy nasz zbiór grup do do słownika w obiekcie przechowującego id grupy oraz listę(grupę) serwerów.

Aby uzyskać unikalne grupy serwerów została zaimplementowana własna lista serwerów gdzie przesłoniliśmy metody Equals oraz GetHashCode dzięki którym możemy porównywać obiekty serwera

|  |
| --- |
| public class ServerList<T> : List<T> , IEnumerable<T>      {          public ServerList()          {          }          public ServerList(List<T> list)          {              foreach(T item in list)              {                  this.Add(item);              }          }          public override bool Equals(object obj)          {              if (obj == null)                  return false;              ServerList<T> list = obj as ServerList<T>;              if (list == null)                  return false;              if (list.Count != this.Count)                  return false;              bool same = true;              this.ForEach(thisItem =>              {                  if (same)                  {                      same = (null != list.FirstOrDefault(item => item.Equals(thisItem)));                  }              });              return same;          }          public override int GetHashCode()          {              unchecked              {                  int hash = 1;                  foreach (var foo in this)                  {                      hash = hash + (foo as Server).GetHashCode() / 2;                  }                  return hash;              }          }      } |

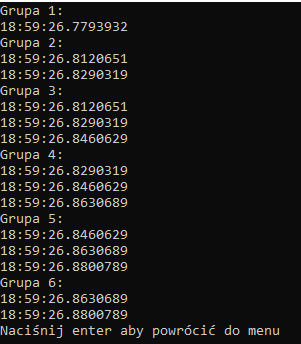
Dodatkowo do zbioru zdefiniowaliśmy własny komparator pozwalający zachować unikalność grup

|  |
| --- |
| public class ServerListComparer : IEqualityComparer<ServerList<Server>>      {          public bool Equals(ServerList<Server> x, ServerList<Server> y)          {              return x.Equals(y);          }          public int GetHashCode(ServerList<Server> obj)          {              return obj.GetHashCode();          }      } |

Po pogrupowaniu dostaniemy komunikat:



Po pogrupowaniu możemy wyświetlić powstałe grupy



Ostatnim elementem działania programu jest przeprowadzenie głosowania i wyświetlenie najbardziej prawdopodobnego czasu na konsoli.

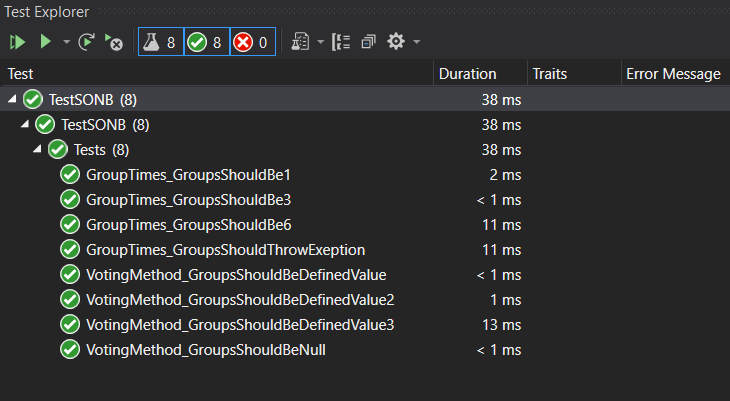
|  |
| --- |
| public DataTime VotingMethod()          {  DataTime time = null;              Dictionary<int, int> bestGroup = new Dictionary<int, int>();              int maxSupport = 0;              List<TimeSpan> times = new List<TimeSpan>();              foreach (KeyValuePair<int, ServerList<Server>> item in groups)              {                  int localMaxSupport = item.Value.Sum(x => x.Weight);                  if(localMaxSupport > maxSupport)                  {                      maxSupport = localMaxSupport;                      bestGroup = new Dictionary<int, int>();                      bestGroup.Add(item.Key, maxSupport);                  }                  else if (localMaxSupport == maxSupport)                  {                      bestGroup.Add(item.Key, maxSupport);                  }              }              if (bestGroup.Count == 1)              {                  ServerList<Server> serverTimes = groups.GetValueOrDefault(bestGroup.FirstOrDefault().Key);                  foreach(Server server in serverTimes)                  {                      for(int i = 0; i < server.Weight; i++)                      {                          times.Add(server.Time.Value.TimeOfDay);                      }                  }                  double doubleAverageTicks = times.Average(timeSpan => timeSpan.Ticks);                  long longAverageTicks = Convert.ToInt64(doubleAverageTicks);                  time = new DataTime(longAverageTicks);              }              else if (bestGroup.Count > 1)              {                  double maxAvgSupport = 0;                  Dictionary<int, ServerList<Server>> groupsLocal = new Dictionary<int, ServerList<Server>>();                  List<int> maxAvgSupportList = new List<int>();                  foreach(KeyValuePair<int, int> item in bestGroup)                  {                      groupsLocal.Add(item.Key, groups.GetValueOrDefault(item.Key));                  }                  foreach (KeyValuePair<int, ServerList<Server>> item in groupsLocal)                  {                      double localAvgMaxSupport = item.Value.Average(x => x.Weight);                      if (localAvgMaxSupport > maxAvgSupport)                      {                          maxAvgSupport = localAvgMaxSupport;                          maxAvgSupportList.Add(item.Key);                      }                      else if (localAvgMaxSupport == maxAvgSupport)                      {                          maxAvgSupportList.Add(item.Key);                      }                  }                  ServerList<Server> serverTimes = groups.GetValueOrDefault(maxAvgSupportList.FirstOrDefault());                  foreach (Server server in serverTimes)                  {                      for (int i = 0; i < server.Weight; i++)                      {                          times.Add(server.Time.Value.TimeOfDay);                      }                  }                  double doubleAverageTicks = times.Average(timeSpan => timeSpan.Ticks);                  long longAverageTicks = Convert.ToInt64(doubleAverageTicks);                  time = new DataTime(longAverageTicks);              }  return time;            } |

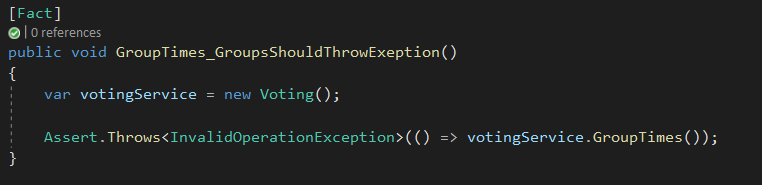
Metoda działa w następujący sposób:   
Tworzymy nowy słownik który będzie przechowywał najlepsze grupy (tj. id grupy oraz jej poparcie) oraz listę przechowująca nasze czasy. Następnie iterujemy po utworzonych wcześniej grupach w celu wyłonienia najlepszej(z największym poparciem). Po zakończeniu tej operacji możemy uzyskać dwie różne drogi którymi możemy podążyć:  
- gdy w słowniku przechowującym najlepsze grupy znajduje się tylko jeden element wyliczamy średnią ważoną czasów znajdujących się w grupie a następnie zwracamy wynik

- gdy w słowniku przechowującym najlepsze grupy znajduje się więcej niż jeden element sprawdzamy dla której grupy istnieje większe średnie poparcie dla pojedynczego elementu w grupie (w przypadku takiego samego średniego poparcia dla pojedynczego elementu grupa wybierana jest losowo). Następnie dla wybranej w ten sposób grupy wyliczamy średnią ważoną czasów znajdujących się w grupie a następnie zwracamy wynik



1. **Testy**



Do aplikacji zostały również dodane testy mające sprawdzić poprawność działania naszej aplikacji. Zostały sprawdzone najważniejsze metody tj. GroupTimes() oraz VotingMethod(). Owe metody zostały przetestowane pod wieloma kątami aby pokryć testami każdą odnogę występująca w metodzie. Wszystkie testy zakończyły się powodzeniem. Poniżej został przedstawiony przykładowy kod testu.  


1. **Podsumowanie**

Dzięki powyższemu projektowi nauczyliśmy się jak zaimplementować metodę głosowania przybliżonego a także dowiedzieliśmy się że możemy go używać gdy wyniki generowane przez bezbłędne moduły systemu się różnią, a źródłem takich niedokładności mogą być np. błędy zaokrąglania w obliczeniach arytmetycznych lub opóźnienia synchronizacyjne w celu znalezienia tego najbardziej prawidłowego wyniku.