Kolejkowanie wiadomości (RabbitMQ)
laboratorium
2022

T. Goluch , K.M. Ocetkiewicz

1. Wstęp

Kolejkowanie wiadomości pozwala na współpracę pomiędzy procesami, które niekoniecznie muszą działać w tym samym czasie. Oznacza to, że jeden z komunikujących się procesów może być przez pewien okres czasu niedostępny. Wiadomości przesyłane pomiędzy takimi procesami, działającymi na tym samym bądź różnych komputerach, przesyłane są z wykorzystaniem kolejek. Krótko mówiąc, jeśli proces do którego ma trafić wiadomość nie jest z jakiś powodów dostępny to może ona na niego poczekać u pośrednika w kolejce. Mechanizm taki podnosi niezawodność budowanych systemów poprzez gwarancję przekazania informacji oraz może poprawić jego wydajność naturalnie wprowadzając model przetwarzania asynchronicznego.

Jedną z dostępnych darmowych i zdobywających coraz większą popularność implementacji pośrednika wiadomości jest RabbitMQ (poniższy rysunek). Napisany został w oparciu o język Erlang, wydany przez firmę Ericson, który został zaprojektowany do wytwarzania systemów rozproszonych wykorzystujących dużą liczbę wątków i charakteryzujących się dużą niezawodnością.

Jest dostępny pod najpopularniejsze systemy operacyjne: Windows. Linux i OS X. Charakteryzuje się bogatą lista wspieranych języków i framework'ów: Java i Spring, .NET, Ruby, Python, PHP, Objective-C i Swift, C, C++ oraz wiele innych. Podczas poznawania tego systemu potrzebna będzie nam znajomość podstawowych terminów:

serwer/pośrednik (*ang. server/broker*) – oprogramowanie RabbitMQ implementujące protokół – Advanced Message Queuing Protocol[1] (AMQP), uruchomione na lokalnym bądź dostępnym zdalnie komputerze.

klient (*ang. producer*) – aplikacja będąca klientem usług dostarczanych przez serwer RabbitMQ.

połączenie (*ang. connection*) – połącznie protokołu warstwy aplikacji AMQP, wykorzystującego protokół TCP w celu niezawodnego dostarczania wiadomości. Połączenia te używają autentykacji i mogą być szyfrowane przy użyciu TLS (SSL).

kanał (*ang. channel*) – jedno połączenie dzielone jest na kanały, co pozwala to na współdzielenie jednego połączenia TCP.

kolejka (*ang. queue*) – bufor przechowywujący wiadomości.

centrala wiadomości (*ang. exchange*) – odbiera wiadomości od producentów i wkłada je do odpowiedniej kolejki/kolejek, w skrajnym przypadku może odrzucić komunikat. To co się stanie z komunikatem zależy od typu wymiany (*ang . exchange type*). Możemy wyróżnić cztery typy wymiany: *direct*, *topic*, *headers*, *fanout*. Ich funkcjonalność i różnice zostaną dokładnie przedstawione później.

wiązanie (*ang. binding*) – związek pomiędzy centralą wiadomości a kolejką.

producent (*ang. producer*) – aplikacja użytkownika (klient) wysyłająca wiadomości. Może ona wysyłać wiadomości do konkretnego albo do grupy konsumentów.

konsument (*ang. consumer*) – aplikacja użytkownika odbierająca wiadomości od producenta.

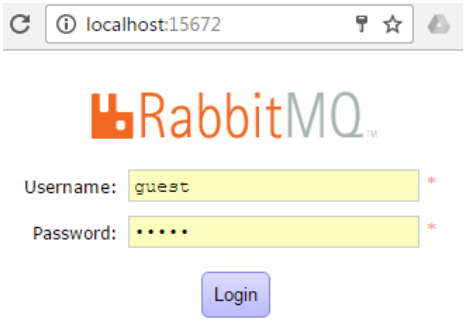
wydawca (*ang. publisher*) – producent w modelu wydawca – subskrybent, wydawca nie przechowuje informacji o tym do kogo mają trafić jego wiadomości.

subskrybent (*ang. subscriber*) – konsument modelu wydawca – subskrybent, zgłasza subskrypcję na konkretny typ komunikatów, nie jest zainteresowany kto jest wydawcą.

[1] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

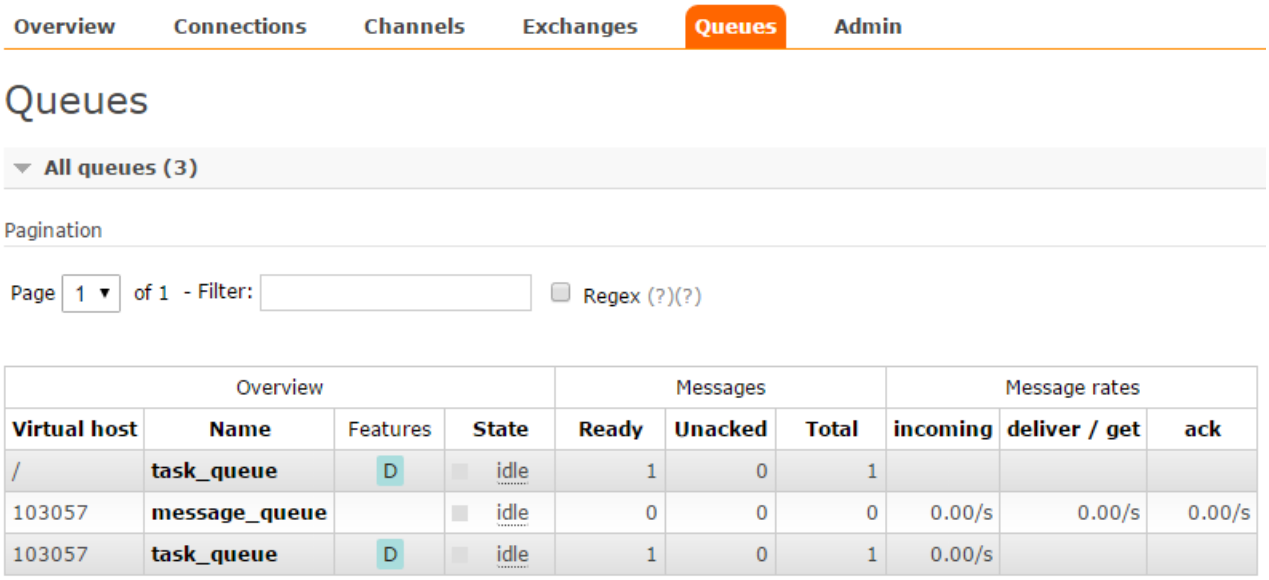
2. Instalacja i administracja serwera

RabbitMQ dostępny jest pod adresem: https://www.rabbitmq.com/download.html i wymaga uprzedniej instalacji języka Erlang, dostępnego jest na stronie: http://www.erlang.org/downloads. W następnym kroku włączyć panel administracyjny dostarczany w formie pluginu. Dokonujemy tego z wiersza poleceń *RabbitMQ Command Prompt (sbin dir)* uruchamianym z menu start i za pomocą polecenia: `rabbitmq-plugins enable rabbitmq_management`. Domyślnie GUI znajduje się pod adresem: http://localhost:15672/ login i hasło to: guest/guest (poniższy rysunek).



Rysunek 1 - Ekran logowania do panelu administracyjnego RabbitMQ.

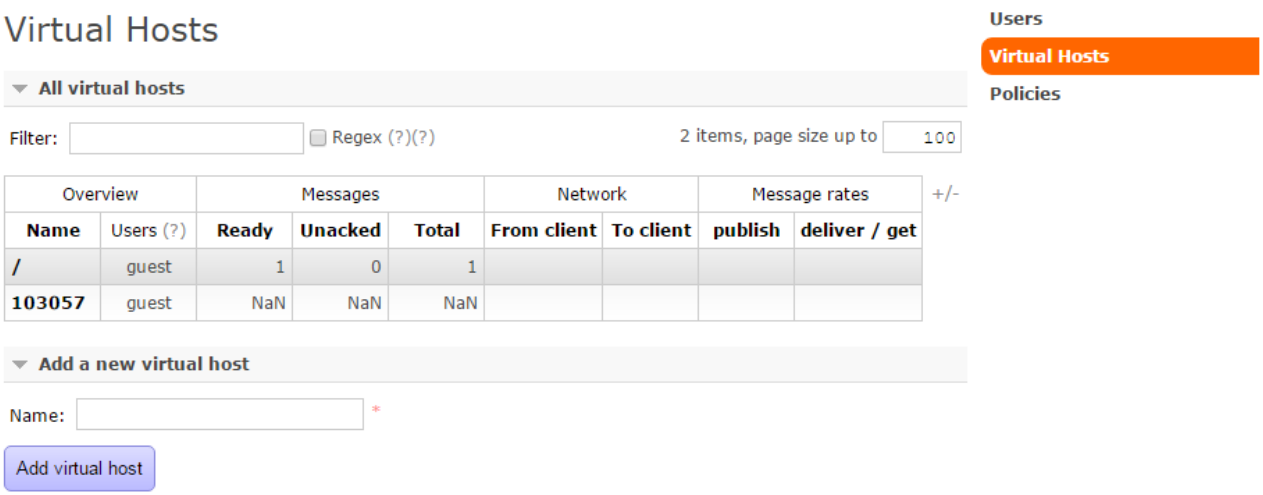
W panelu administracyjnym możemy przeglądać aktywne połączenia oraz kanały, a także dodawać, usuwać, zarządzać i sprawdzać stan kolejek, central wiadomości (poniższy rysunek).



Rysunek 2 - lista kolejek serwera.

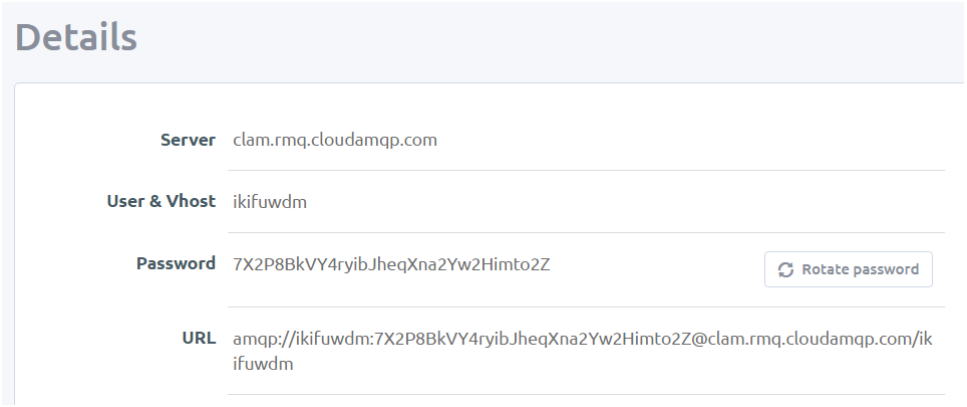
wirtualny host

W menu *Admin* panelu administracyjnego po prawej stronie ekranu mamy zakładkę *Virtual Hosts*. Pozwala ona na łatwe dodawanie, zarządzanie i usuwanie tzw. hostów wirtualnych (poniższy rysunek). Pozwalają ona na oddzielenie aplikacji używających tej samej instancji RabbitMQ. Ci sami użytkownicy mogą mieć różne prawa dostępu na różnych hostach. Kolejki i centrale wiadomości przypisywane są do konkretnego hosta co ogranicza prawdopodobieństwo powstania kolizji nazw.



Rysunek 3 - Lista wirtualnych hostów w panelu administracyjnym.

Alternatywą do uruchamiania własnego serwera jest skorzystanie z RabbitMQ udostępnianego w formie serwisu – https://www.cloudamqp.com/. Dostępny jest plan darmowy deweloperski obsługujący do 1MB wiadomości miesięcznie, 20 równoległych połączeń, 100 kolejek, 10.000 zakolejkowanych wiadomości. Maksymalny okres bezczynności wynosi 28 dni. Po założeniu konta i zalogowaniu do *CloudAMPQ* informacje potrzebne do utworzenia fabryki połączeń dostępne są pod adresem: https://customer.cloudamqp.com/instance - należy wybrać interesującą na instancję (poniższy rysunek).



Rysunek 4 - Informacje potrzebne dla fabryki połączeń.

Utworzenie fabryki dla powyższych wartości wyglądało by następująco:

```
var factory = new ConnectionFactory()
{
    UserName = "ikifuwdm",
    Password = "<HASŁO>",
    HostName = "clam.rmq.cloudamqp.com",
    VirtualHost = "ikifuwdm"
};
```

3. Klient[2], połączenie, kanał i kolejka

Jak wspomniano wcześniej RabbitMQ wspiera wiele framework'ów oraz powiązanych z mini języków programowania. Oczywiście dotyczy to również języka C# i środowiska .NET. Klienta RabbitMQ można łatwo dodać za pomocą Nuget'a. wystarczy w oknie menedżera pakietów (Package Manager Console) wydać następujące polecenie:

[2] https://www.rabbitmq.com/dotnet-api-guide.html

```
Install-Package RabbitMQ.Client
```

jeśli docelowy projekt nie jest domyślnym, należy jeszcze jawnie go określić:

```
Install-Package RabbitMQ.Client -Project <nazwa-projektu>.
```

Do poprawnego działania najnowszej wersji klienta RabbitMQ dostępnej poprzez NuGet'a (wersja 4.1.x) wymagany jest framework .NET w wersji przynajmniej 4.5.1.

Obiekt klasy `ConnectionFactory` pozwala na utworzenie połączenia protokołu AMQP[3] (obiektu implementującego `IConnection`) za pomocą metody `CreateConnection()`. Z kolei metoda `CreateModel()` interfejsu `IConnection` pozwala na utworzenie kanału (obiekt implementujący `IModel`) dostarczającego większość interesujących nas operacji. Zaleca się użycie obiektów reprezentujących połącznie i kanał wewnątrz klauzuli `using`. Pozwala to na automatycznie zwolnienie zasobów w momencie gdy sterowanie opuszcza blok kodu nawet w przypadku nieobsłużonego wyjątku. W przeciwnym przypadku należy pamiętać o ich zamknięciu jawnie wywołując metody: `Close()` oraz `Dispose()`.

Do przechowywania wiadomości służą kolejki i charakteryzują się one, teoretycznie nieograniczoną pojemnością. Metoda `QueueDeclare()` interfejsu `IModel` pozwala na ich utworzenie. Jej działanie jest idempotentne, kolejka zostanie utworzona tylko jeżeli nie istnieje, zatem możemy ją zadeklarować w dowolnie wielu projektach. Przyjmuje nast. parametry:

- `string` **`queue`** – nazwa kolejki, jeśli nie podamy nazwy zostanie wygenerowana nazwa domyślna z przedrostkiem: `amq.gen-*`. W przypadku próby połączenia się z kolejką bez wyspecyfikowanej nazwy połączymy się z ostatnią zadeklarowaną na tym kanale.
- `boolean` **`durable`** – czy kolejka ma być trwała?
- `boolean` **`exclusive`** – czy kolejka ma być wykorzystywana przez tylko jedno połączenie i zostać usunięta po jego zamknięciu?
- `boolean` **`autoDelete`** – czy kolejka ma zostać usunięta po odczytaniu ostatniej wiadomości?
- `IDictionary<String, object>` **`arguments`** `= null` – (opcjonalny) używany przez pośredników do realizacji dodatkowych funkcji, takich jak TTL wiadomości itp.

Przykładowy kod klienta:

```
var factory = new ConnectionFactory()
{
    UserName = "guest",
    Password = "guest",
    HostName = "localhost",
    VirtualHost = "103057"
};
using (var connection = factory.CreateConnection())
using (var channel = connection.CreateModel())
{
      channel.QueueDeclare("message_queue", false, false, false, null);
      // send/receive message
}
```

Należy pamiętać, że po opuszczeniu klauzuli `using` komunikacja będzie przerwana, jeśli chcemy poczekać z zakończeniem procesu (np. komendą `ReadKey`) to należy to zrobić wewnątrz klauzuli jeszcze przed zwolnieniem zasobów `channel` i `connection`.

4. Komunikacja nadawca (producent) – odbiorca (konsument)

Jak już wspomnieliśmy producent to program wysyłający wiadomości, a konsument to ich odbiorca. Wielu producentów jak i konsumentów może korzystać z jednej kolejki. Oczywiście producent, konsument jak i pośrednik nie muszą, działać na tej samej maszynie ale na potrzeby laboratorium ograniczymy się do jednego stanowiska

[3] Domyślanie do komunikacji wykorzystywany jest protokół Advanced Message Queuing Protocol (AMQP), aktualnie w wersji 0-9-1. Do wersji 1.0 można wykorzystać eksperymentalny plugin - https://github.com/rabbitmq/rabbitmq-amqp1.0.

komputerowego. W modelu nadawca-odbiorca, obydwie aplikacje klienckie (producent i odbiorca) będą jawnie specyfikować do/z jakiej kolejki będą trafiać/pobierać komunikaty.

konsument

Metoda `BasicConsume()` interfejsu `IModel` pozwala na odbieranie wiadomości. Przyjmuje nast. parametry:

- `string` **`queue`** – nazwa kolejki, istotna jeżeli chcemy aby wiadomości były wysyłane i odbierane dokładnie z tej kolejki.
- `boolean` **`noAck`** – domyślna wartość `false` powoduje, że RabbitMQ będzie oczekiwał potwierdzenia o poprawnym odebraniu i przetworzeniu wiadomości, w innym przypadku nastąpi jej ponowienie.
- `IBasicConsumer` **`consumer`** – obiekt konsumenta może być to:
  - klasa dziedzicząca po `DefaultBasicConsumer` dostęp do wiadomości uzyskujemy wewnątrz przeciążonej metody `HandleBasicDeliver`,
  - obiekt klasy `EventingBasicConsumer`, należy się podpiąć do zdarzenia `Received`,

Przykładowy kod odbierający wiadomość bezpośrednio z kolejki (obiekt `consumer` dziedziczy po klasie `DefaultBasicConsumer`):

```
consumer = new MyConsumer(channel);
channel.BasicConsume("message_queue", true, consumer);
```

Kod klasy `DefaultBasicConsumer`:

```
class MyConsumer : DefaultBasicConsumer
{
    public MyConsumer(IModel model) : base(model) { }

    public override void HandleBasicDeliver(string consumerTag, ulong deliveryTag, bool redelivered, string exchange, string routingKey, IBasicProperties properties, ReadOnlyMemory<byte> body)
    {
        var message = Encoding.UTF8.GetString(body.ToArray());
        // show message
    }
}
```

To samo z podpięciem do zdarzenia `Received` obiektu klasy `EventingBasicConsumer` (wersja ze zwykłą metodą):

```
consumer.Received += Receiv;
      ...

private static void Receiv(object model, BasicDeliverEventArgs ea)
{
    var body = ea.Body;
    var message = Encoding.UTF8.GetString(body);
    ConsoleCol.WriteLine(" [x] Received:'{0}'", message, ConsoleColor.Yellow);
}
```

i z wyrażeniem lambda:

```
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body;
    var message = Encoding.UTF8.GetString(body);
    // show message
};
channel.BasicConsume("message_queue", true, consumer);
```

Proszę zwrócić uwagę, że powyższe metody jednynie rejestrują obiekty czy metody, które obsłużą wiadomość, gdy ta zostanie dostarczona. W żadnym z tych przypadków program nie czeka na odebranie wiadomości.

producent

Do wysyłania wiadomości służy metoda `BasicPublish()` interfejsu `IModel`, przyjmuje następujące parametry:

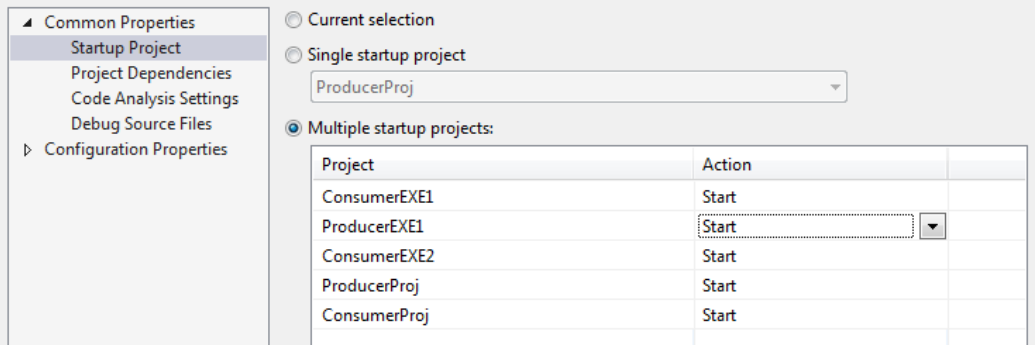
- `string` **`exchange`** – jeśli jest to pusty napis to pozwala na dokładne określenie w parametrze routingKey do której kolejki chcemy przekazać wiadomość (opcja wykorzystywana w komunikacji nadawca-odbiorca).
- `string` **`routingKey`** – służy do routingu komunikatów w zależności od rodzaju parametru `exchange` i konfiguracji, jak opisano wcześniej dla pustego napisu powinna być to nazwa kolejki.
- `boolean` **`mandatory`** (opcjonalny) – czy serwer ma informować nadawcę o problemie z przekazaniem wiadomości do kolejki, czy tylko ją usunąć?
- `AMQP.BasicProperties` **`props`** – obiekt implementujący interfejs `IBasicProperties` określa właściwości wiadomości takie jak: typ i kodowanie MIME zawartości, priorytet (0-9), Id użytkownika/wiadomości, lista nagłówków, itp., Właściwość `Persistance` określa tryb dostarczenia (nietrwały/trwały), w przypadku tego drugiego należy pamiętać, że zadziała on jedynie z kolejkami trwałymi, tj. takimi których parametr **`durable`** podczas deklaracji kolejki ustawiony był na `true`. Jeśli zadawalają nas domyślne ustawienia to może być `null`.
- `Byte[]` **`body`** – treść wiadomości.

Przykładowy kod wysyłający wiadomość bezpośrednio do kolejki:

```
string message = "my message";
var body = Encoding.UTF8.GetBytes(message);
channel.BasicPublish("", "message_queue", null, body);
```

Uruchamianie jednocześnie wielu instancji jednego projektu

Aby sprawdzić możliwość współpracy dowolnej liczby producentów i konsumentów z jedną kolejką będziemy musieli uruchamiać klika procesów jednocześnie. Można uruchomić Visual Studio tak aby podczas debugowania uruchamiało kilka projektów jednocześnie np. producenta i konsumenta. Służy do tego zakładka *Common Properties → Startup Project* we właściwościach solucji.



Rysunek 5 - Domyślne startowanie wielu projektów.

Jeśli chcemy uruchamiać kilka instancji tego samego projektu (np. kilku konsumentów) to można to zrobić przynajmniej na trzy sposoby. Niestety żaden nie jest doskonały:

1. Dla każdej instancji stworzyć osobny projekt konsolowy i w ustawieniach w menu *Debug* projektu ustawić opcję *Start Action* na *Start external program:* podając ścieżkę do pliku wykonywalnego, którego dodatkową instancję zamierzamy uruchomić. Nie powinniśmy usuwać automatycznie wygenerowanego pliku *Program.cs* ponieważ podczas uruchamiania debbugera otrzymamy ostrzeżenie, że program nie może odszukać metody `Main`.

2. Można również dodać do solucji za pomocą *Add → Existing Project…* bezpośrednio plik wykonywalny *.exe* znajdujący się w folderze *Debug*. Możliwe, że wcześniej będzie wymagana zmiana nazwy projektu którego binarka jest właśnie dodawana (domyślnie nazwa projektu jest identyczna jak gotowego podzespołu). Jeśli dodajemy kilka binarek, dodając kolejną dostaniemy komunikat, że taki projekt o takiej nazwie już istnieje. Można chwilowo oszukać Visual

Studio. W tym celu należy zmienić nazwę projektu (F2) wcześniej dodanej binarki na dowolną inną. Pomimo iż zmiana nie zostanie wyświetlona, będzie można dodać kolejny raz naszą binarkę (może się okazać, że została usunięta – należy wtedy wykonać *Rebuild* na odpowiadającym jej projekcie z kodem). Niestety po zamknięciu i przy ponownym wczytywaniu solucji Visual Studio zorientuje się, że mamy projekty o identycznej nazwie i ten opisany zabieg trzeba będzie za każdym razem powtórzyć.

3. Możemy we właściwościach naszego projektu w ustawieniach menu *Debug* ustawić opcję *Start Action* na *Start external program:* podając ścieżkę do wiersza poleceń: *C:\Windows\System32\cmd.exe*. Natomiast w polu *Command line arguments* jako argumenty podać dowolną liczbę instancji naszego programu do uruchomienia: */C "start <nazwa_binarki>.exe & start <nazwa_binarki>.exe …"*. Niestety w tym trybie nie będziemy mogli debugować projektu, aczkolwiek możemy łatwo przełączać się pomiędzy trybem normalnym zaznaczając z powrotem opcję w ustawieniach menu *Debug → Start Project*.

Kolorowanie strumienia wyjściowego konsoli

W celu łatwego rozróżnienia procesów producenta i konsumenta można ustawić im inne kolory, np. wyświetlanej czcionki. Służy do tego komenda: `Console.ForegroundColor = ConsoleColor.<wybrany_kolor>`. Jeśli dodatkowo będziemy chcieli odróżnić komunikaty wypisywane po odebraniu wiadomości to ze względu na wielowątkowy charakter klientów lepszym rozwiązaniem może okazać się wykorzystanie klasy `ConsoleCol` dołączonej w zasobach do instrukcji. Udostępnia ona metody `Write i WriteLine` opakowujące ich odpowiedniki z `System.Console` pozwalające na chwilową zmianę koloru. Pozwoli to na poprawna obsługę sytuacji (wydrukowanie każdego tekstu w odpowiednik kolorze), kiedy główny proces konsoli będzie wypisywał informacje razem z wątkiem obsługującym odebraną wiadomość.

5. Nagłówki

Do wysyłanej wiadomości można dołączyć dowolną liczbę nagłówków. Służy do tego właściwość `Headers` obiektu implementującego interfejs `IBasicProperties`. Jest to właściwość typu `IDictionary<string, object>`, zatem każdy nagłówek to para klucz-wartość, gdzie klucz jest napisem a wartość może być dowolnym obiektem. Obiekt implementujący `IBasicProperties` przekazujemy do metody `BasicPublish` producenta.

```
IBasicProperties properties = channel.CreateBasicProperties();
properties.Headers = new Dictionary<string, object>();
properties.Headers.Add("job sec", 10);
channel.BasicPublish("", "task_queue", properties, body);
```

Po stronie konsumenta jest on dostępny jako parametr w metodzie `HandleBasicDeliver`, bądź w metodzie `BasicDeliverEventArgs` podpinanej do zdarzenia `Received`.

```
consumer.Received += (model, ea) =>
{
    int jobSec = (int)ea.BasicProperties.Headers["job sec"];
    // message processing
}
```

6. Niezawodność komunikacji

utrwalanie wiadomości

W punkcie opisującym tworzenie kolejek zaznaczyliśmy, że parametr **`durable`** metody `QueueDeclare()` ustalony na `true` powoduje utworzenie kolejki trwałej. W panelu administracyjnym jest to sygnalizowane pojawieniem się oknie *Overwiew→Features* literki D na niebieskim tle. Należy pamiętać, że nie można zadeklarować kolejki trwałej jeśli istnieje już kolejka nietrwała o takiej samej nazwie. Taka kolejka może przetrwać restart pośrednika/serwera (upadek systemu albo usługi, można zasymulować komendami: `rabbitmq-service.bat stop/start`[4])? Można zauważyć,

[4] Skrypt `rabbitmq-service.bat` znajduje się w podkatalogu `sbin` katalogu zawierającego serwer RabbitMQ, ponadto do jego uruchomienia wymagane są uprawnienia administratora. W menu start powinny być dostępne skróty z polecaniami pozwalającymi na zatrzymanie i uruchomienie serwera, one również wymagają uprawnień administratora.

że przetrwają tylko wiadomości których właściwość `Persistent` będzie ustawiona na `true`. Taka wiadomość w przypadku kiedy pośrednik nie może od razu przekazać jej dalej zostanie utrwalana na dysku. Przykładowy kod wysyłający wiadomości trwałe:

```
IBasicProperties prop = channel.CreateBasicProperties();
prop.Persistent = true;
channel.BasicPublish("", "<nazwa_kolejki>", prop, body);
```

potwierdzenia

Wyobraźmy sobie sytuację, że wiadomość została poprawnie przesłana do konsumenta jednak ten zakończył swój żywot i nie zdążył poprawnie przetworzyć otrzymanej wiadomości. Aby się zabezpieczyć przed takimi przypadkami możemy wykorzystać mechanizm potwierdzeń. W celu wymuszenia potwierdzania wiadomości przez konsumentów musimy odpowiednio zmodyfikować ustawienia QoS (Quality of service). Funkcja `BasicQos()` interfejsu IModel pobiera trzy parametry:

- `uint prefetchSize` – ograniczenie na liczbę bajtów wiadomości, 0 – brak ograniczenia,
- `ushort prefetchCount` – maksymalna liczba wiadomości po której wymagane będzie powiadomienie, 0 – nielimitowana, użyteczne w celu równoważenia obciążenia; pośrednik nie będzie wysyłał więcej zadań do przetworzenia dopóki nie otrzyma potwierdzenia o wykonaniu wcześniej wysłanych,
- `bool global` – `true` jeśli ustawienia dotyczą całego kanału a nie tylko bieżącego konsumenta,

Funkcję `BasicQos()` wywołujemy w procesie konsumenta. Do potwierdzania służy metoda `BasicAck()` interfejsu `IModel` za jej pomocą można potwierdzać naraz jedną bądź więcej wiadomości, przyjmuje parametry:

- `ulong deliveryTag` – tag przekazywany jako parametr do metody obsługującej wiadomość, reprezentujący przetwarzaną wiadomość.
- `bool multiple` – jeśli `false` to potwierdzenie dotyczy tylko wiadomości dostarczonej z przekazanym jako pierwszy parametr z `deliveryTag`, dla `true` dodatkowo potwierdzane są wszystkie wcześniejsze od niej wiadomości.

transakcje

Wysyłane komunikaty mogą uczestniczyć w transakcjach, jednak należy pamiętać aby ich nie nadużywać ponieważ potrafią skutecznie spowolnić, a nawet zablokować działanie całego systemu. Do przełączenia kanału w tryb transakcyjny służy metoda `TxSelect()` interfejsu `IModel`. Następnie możemy już wysyłać dowolną liczbę wiadomości w ramach jednej transakcji. Wywołanie metody `TxCommit()` na tym samym kanale (ten sam obiekt) spowoduje zatwierdzenie a wywołanie metody `TxRollback()` odrzucenie wszystkich wysłanych wiadomości i potwierdzeń. W obydwu przypadkach od razu rozpoczęta zostaje nowa transakcja. Zauważmy, że w przypadku rollback'a niepotwierdzone wiadomości nie zostaną ponownie wysłane z automatu, jeśli jest to wymagane musimy to zrobić ręcznie.

```
channel.TxSelect();
// send messages
if (/*OK*/) channel.TxCommit();
else channel.TxRollback();
```

W poprzedniej instrukcji – dotyczącej WCF – opisano interfejs `IEnlistmentNotification` pozwalający na utworzenie zasobu obsługiwanego przez MSDTC (Microsoft Distributed Transaction Coordinator). Do implementacji należy wykorzystać opisane metody: `TxCommit()`, `TxSelect()` i `TxRollback()` co pozwoli na automatyczną rejestrację w transakcji realizowanej przez oraz protokół dwufazowego zatwierdzania.

7. Realizacja wzorca (Competing Consumers[5])

Wykorzystując komunikację nadawca-odbiorca możemy łatwo zbudować system potrafiący równomiernie rozdzielać długotrwałe zadania pomiędzy wiele procesów zajmujących się ich przetwarzaniem. Wystarczy utożsamić wiadomości z zadaniami. Jeden bądź więcej nadawców (producentów) będzie rozdysponowywać zadania wysyłając wiadomości do odpowiedniej kolejki. Dowolna liczba odbiorców (konsumentów) będzie przetwarzać zadania otrzymane w formie komunikatów. Jeżeli wykorzystamy domyślne ustawienia QoS (brak wymaganego potwierdzania przetworzenia

[5] http://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html

wiadomości/zadania) to wszystkie komunikaty/zadania zostaną równomiernie rozdysponowane pomiędzy dostępnych klientów. Oznacza to, że jeśli w międzyczasie dołączy nowy klient to nie miał nic do roboty, nawet jeśli pozostali konsumenci maja jeszcze sporo nieprzetworzonych komunikatów/zadań. Innym przykładem problemu jaki można napotkać jest możliwość istnienia zadań o zróżnicowanym stopniu złożoności, jeśli jeden z procesów niefortunnie dostanie zadanie/zadania wymagające długotrwałych obliczeń, albo jest po prostu mniej wydajny, to nie będzie możliwości aby inny konsument mu pomógł. Aby łatwo uchronić się przed takimi scenariuszami, należy w kodzie klienta zmienić ustawienia QoS tak aby wymagane było wysłanie potwierdzenia przetworzenia dla każdej wiadomości/zadania. W ten sposób aktualnie nieprzetwarzane komunikaty/zadania będą czekały w kolejce na pierwszego klienta który oznajmi, że jest wolny – wysyłając potwierdzenie za pomocą metody `BasicConsume()`.

8. Wołania zwrotne

Podczas wykonywania zadania przez konsumenta możemy sobie zażyczyć zwrotu wyniku wykonywanych obliczeń. Takie rozwiązanie to nic innego jak zdalne wywołanie procedury (*ang. Remote Procedure Call – RPC*). RabbitMQ również obsługuje ten wzorzec. Wystarczy zadeklarować kolejkę na której będziemy oczekiwali odpowiedzi:

```
        // consume response from consumer
        string replyQueueName = channel.QueueDeclare().QueueName;
        EventingBasicConsumer consumer = new EventingBasicConsumer(channel);
        channel.BasicConsume(replyQueueName, true, consumer);
```

Aby konsument wiedział do kogo odesłać odpowiedź musimy podać nazwę właśnie utworzonej tymczasowej kolejki we właściwości `ReplyTo` parametru `properties` implementującego `IBasicProperties`. Dodatkowo aby odnaleźć naszą odpowiedź wśród innych, które mogą się pojawić w kolejce, należy jej nadać unikatowy identyfikator globalny GUID (`"CorrelationId"`):

```
        // required for response
        properties.ReplyTo = replyQueueName;
        var corrId = Guid.NewGuid().ToString();
        properties.CorrelationId = corrId;
```

Teraz pozostaje nam jedynie czekać na odpowiedź:

```
consumer.Received += (model, ea) =>
{
    if (ea.BasicProperties.CorrelationId == corrId)
    {
        ConsoleCol.Write("\n{0}\n", Encoding.UTF8.GetString(ea.Body), ConsoleColor.Blue);
    }
};
```

Wysłanie informacji zwrotnej w postaci czasu zakończenia obliczeń mogło by wyglądać następująco (przypadek z implementacją obiektu dziedziczącego po klasie `DefaultBasicConsumer`):

```
        var responseBytes = Encoding.UTF8.GetBytes(DateTime.Now.ToLongTimeString());
        var replyProps = Model.CreateBasicProperties();
        replyProps.CorrelationId = properties.CorrelationId;
        Model.BasicPublish("", properties.ReplyTo, replyProps,  responseBytes);
```

9. Komunikacja wydawca-subskrybent

Wzorzec wydawca-subskrybent pozwala publikować wiadomości które będą trafiać tylko do zainteresowanych nimi subskrybentów. W celu realizacji tego scenariusza teraz wiadomości nie są przekazywane bezpośrednio do kolejki tylko do centrali wiadomości. Centrala wiadomości wie jakie wiadomości powinny trafić do jakich kolejek. Metoda `ExchangeDeclare()` interfejsu `IModel`, tworzy – analogicznie jak to miało miejsce w przypadku kolejek – w sposób idempotentny centralę wiadomości. Co oznacza, że możemy ją zadeklarować zarówno w procesach wydawców jak i subskrybentów. Przyjmuje następujące parametry:

- `string exchange` – nazwa centrali wiadomości.
- `string type` – typ centrali: `direct`, `topic`, `headers`, `fanout`.
- `bool durable = false` – analigicznie jak w przypadku kolejki.
- `bool autoDelete = false` – analigicznie jak w przypadku kolejki.
- `IDictionary<string, object> arguments = null` – (opcjonalny) analogicznie jak w przypadku kolejki.

subskrybent

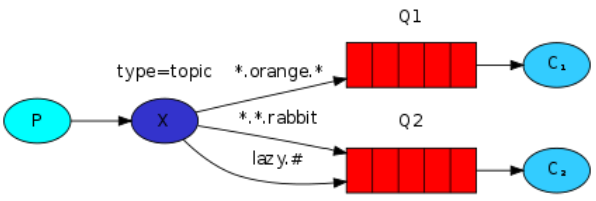
W procesie subskrybenta należy powiązać zadeklarowane centrale wiadomości z kolejkami. Ponieważ nazwy kolejek nie będą istotne dla wydawców możemy wykorzystać tzw. kolejki anonimowe. Służy do tego polecenie:

```
var queueName = channel.QueueDeclare().QueueName;
```

Zmienna `queueName` przechowuje wygenerowaną automatycznie nazwę kolejki w formacie `"amq.gen-*"`, którą będziemy wykorzystywać w następnym kroku podczas wiązania jej z central wiadomości. Takie wiązanie realizujemy za pomocą metody `QueueBind()`, przyjmuje ona nast. parametry:

- `string queue` – nazwa kolejki.
- `string exchange` – nazwa centrali wiadomości.
- `string routingKey` – jego znaczenie zależy od typu centrali wiadomości:
  - **`fanout`** – parametr jest ignorowany, wszystkie wiadomości przesłane do centrali będą przekazywane do powiązanych kolejek.
  - **`direct`** – tylko wiadomości przesłane do centrali posiadające identyczną wartość parametru **`routingKey`** zostaną włożone do powiązanej kolejki.
  - **`topic`** – jak w przypadku `direct` z tą różnicą, że teraz parametr **`routingKey`** wiadomości przesyłanych do centrali składa się z dowolnych słów (przeważnie określających pewne cechy) połączonych kropkami. Wielkość klucza ograniczona jest do 255 bajtów. Deklarując **`routingKey`** wiązania możemy używać dwóch znaków specjalnych:
    - * (gwiazdka) zastępuje dokładnie jedno słowo,
    - # (hash) zastępuje zero bądź więcej słów.

    Pozwala to na filtrowanie interesujących na wiadomości wg. konkretnych cech. Przykładowo dla klucza o budowie: `"<celerity>.<colour>.<species>"` określającego takie cechy zwierząt jak prędkość, kolor i gatunek. Łatwo można powiązać dwie kolejki, do pierwszej mają trafiać komunikaty o dowolnych pomarańczowych zwierzętach, a do drugiej o królikach i wszystkich powolnych zwierzętach (poniższy rysunek). Zatem wiadomość o kluczu `"quick.orange.fox"` trafi tylko do pierwszej kolejki a `"lazy.brown.fox"` tylko do drugiej. Wiadomości `"quick.orange.rabbit"` i `"lazy.pink.rabbit"` trafią do obydwu kolejek dokładnie raz mimo, że druga z nich pasuje do obydwu kluczy drugiej kolejki. Natomiast wiadomość z kluczem `"quick.brown.fox"` zostanie zignorowana (nie trafi do żadniej kolejki).
  - **`headers`** – jak w przypadku `direct` z tą różnicą, że rolę kluczy routingu spełniają teraz nagłówki, a `routingKey` jest ignorowany. Mogą one przyjmować dowolne typy danych, a nie – jak to miało miejsce wcześniej tylko `string`. Nagłówek specjalny o kluczu `x-match` publikowanej wiadomości może przyjąć dwie wartości:
    - `any` – wystarczy jedno dopasowanie klucza w wiązaniu kolejki z centralą aby wiadomość trafiła do tej kolejki,
    - `all` – wiadomości trafią jedynie do kolejek w których wszystkie klucze spełniają dopasowanie.



Rysunek 6 - Centrala wiadomości typu "topic" (źródło: https://www.rabbitmq.com)

Przykładowy kod wiążący centralę typu `fanout` z kolejką anonimową:

```
channel.ExchangeDeclare("brodcast", "fanout");
var queueName = channel.QueueDeclare().QueueName;
channel.QueueBind(queueName, "brodcast", "");
```

Przykładowy kod wiążący centralę typu `direct` z kolejką anonimową (wartość klucza = `"routingkey1"`):

```
channel.ExchangeDeclare("dir_rout", "direct");
var queueName = channel.QueueDeclare().QueueName;
channel.QueueBind(queueName, "dir_rout", "routingkey1");
```

Przykładowy wiążący centralę typu `topic` z kolejką anonimową (wartość klucza = `"lazy.pink.rabbit"`):

```
channel.ExchangeDeclare("top_rout", "topic");
var queueName = channel.QueueDeclare().QueueName;
channel.QueueBind(queueName, "top_rout", "lazy.pink.rabbit");
```

Przykładowy kod wiążący centralę typu `headers` o wartości nagłówka specjalnego `"x-match"="all"` z kolejką anonimową:

```
channel.ExchangeDeclare("head_rout", " headers");
var queueName = channel.QueueDeclare().QueueName;
var headers = new Dictionary<string, object>{
    { "x-match", "all" }, //any or all
    { "part", "first" },
    { "number", 42 }
};
channel.QueueBind(queueName, "head_rout", "", headers);
```

Konsumpcja komunikatów odbywa się analogicznie jak w przypadku komunikacji nadawca-odbiorca.

wydawca

W procesie wydawcy nie tworzymy kolejek oraz wiązań, a wiadomości publikujemy wprost do centrali wiadomości.

Przykładowy kod wysyłający wiadomość do centrali typu `fanout`:

```
channel.ExchangeDeclare("brodcast", "fanout");
string message = ... // add message
channel.BasicPublish("brodcast", "", null, body);
```

Przykładowy kod wysyłający wiadomość z wartością klucza = `"routingkey1"`, do centrali typu `direct`:

```
channel.ExchangeDeclare("dir_rout", "direct");
string message = ... // add message
channel.BasicPublish("dir_rout", "routingkey1", null, body);
```

Przykładowy kod wysyłający wiadomość z wartością klucza = `"lazy.pink.rabbit"`, do centrali typu `topic`:

```
channel.ExchangeDeclare("top_rout", "topic");
string message = ... // add message
channel.BasicPublish("top_rout", "lazy.pink.rabbit", null, body);
```

Przykładowy kod wysyłający wiadomość do centrali typu `headers`:

```
BasicProperties props = new BasicProperties();
props.Headers = new Dictionary<string, object>{
    { "part", "first" },
    { "number", 42 }
};
channel.ExchangeDeclare("head_rout", "headers");
string message = ... // add message
channel.BasicPublish("head_rout", "", props, body);
```

10. WCF (brak wsparcia od wersji 3.6.9)

Nic nie stoi na przeszkodzie aby jako protokół komunikacyjny (binding) do usługi WCF wykorzystać RabbitMQ. Możemy korzystać z metod jednokierunkowych typu "Fire and Forget" jak i z typowych metod dwukierunkowych pozwalających na zwrócenie wyniku "Request/Reply". Do procesu hostującego i klienta należy za pomocą Nuget'a. dodać bibliotekę *RabbitMQ.ServiceModel.dll*. Niestety wersja 3.6.9 to ostatnia wspierana wersja biblioteki. Do procesu hostującego:

```
Install-Package RabbitMQ.ServiceModel -Project <nazwa-projektu>.
```

Oraz utworzyć hosta i dodać endpoint'a nasłuchującego pod adresem protokołu `amqp`[6]:

```
ServiceHost sh = new ServiceHost(typeof(<klasa_serwisu>),
    new Uri[] { new Uri("http://localhost:1100"), new Uri("soap.amqp:///") });
sh.AddServiceEndpoint(typeof(<interfejs_serwisu>),
    new RabbitMQBinding("localhost", AmqpTcpEndpoint.UseDefaultPort,
    "guest", "guest", "/", 8192, Protocols.AMQP_0_9_1),
    "<adres_serwisu>");
```

Niestety po RabbitMQ nie możemy wystawić endpointa z meta danymi, musimy zrobić to w sposób tradycyjny (np. po protokole http – adres bazowy dla tego protokołu został już podany podczas tworzenia hosta). Po tych czynnościach możemy przejść do tworzenia klienta. Należy dodać webreferencję do serwisu, (powinien on być uruchomiony), co pozwoli nam na łatwe utworzenie fabryki kanałów:

```
var fact = new ChannelFactory<<interfejs_serwisu>>(
    new RabbitMQBinding("localhost", AmqpTcpEndpoint.UseDefaultPort,
    "guest", "guest", "/", 8192, Protocols.AMQP_0_9_1),
    new EndpointAddress("soap.amqp:///<adres_serwisu>"));
```

należy pamiętać, że obiekt `RabbitMQBinding` dostępy jest w przestrzeni nazw `RabbitMQ.ServiceModel`, a obiekty `AmqpTcpEndpoint` i `Protocols` w `RabbitMQ.Client`.

11. Błędy

RabbitMQ.Client.Exceptions.AlreadyClosedException: 'Already closed: The AMQP operation was interrupted: AMQP close-reason, initiated by Peer, code=406, text='PRECONDITION_FAILED - unknown delivery tag 1', classId=60, methodId=80' – **Błąd podczas wołania metody Model.BasicAck(deliveryTag, false). Spowodowny jest przez podanie argumentu boolean noAck = true w metodzie BasicConsume(name_queue, true, consumer);** .

An unhandled exception of type 'RabbitMQ.Client.Exceptions.BrokerUnreachableException' occurred in RabbitMQ.Client.dll Additional information: None of the specified endpoints were reachable – **brak zainstalowanego serwera RabitMQ**.

Plugin configuration unchanged. Applying plugin configuration to rabbit@<nazwa_hosta>... failed. – **plugin został już wcześniej włączony, można to sprawdzić za pomocą polecenia `rabbitmq-plugins.bat list`.**

System.MissingFieldException: „Nie odnaleziono pola: 'RabbitMQ.Client.ConnectionFactory.Protocol'." – **zbyt późna wersja biblioteki RabbitMQ .NET client, proszę skorzystać z wersji ≤ 3.6.9.**

Lektura uzupełniająca:

- https://www.rabbitmq.com/getstarted.html (Tutorials)
- https://www.rabbitmq.com/documentation.html (Documentation)
- https://www.rabbitmq.com/dotnet-api-guide.html (.NET/C# Client API Guide)

[6]Proszę wykorzystać bibliotekę RabbitMQ .NET client w wersji 3.6.9. Późniejsze wersje powodują powstanie wyjątku **System.MissingFieldException:** „Nie odnaleziono pola: 'RabbitMQ.Client.ConnectionFactory.Protocol'."