

# **Core C# and .NET**

by

**Stephen C. Perry**

*Wyłącznie jako materiał porównawczy w ramach PZR 420 i 380*

przez technologię Remoting wymagają, aby przed rozpoczęciem komunikacji obiektu w odrębnych domenach aplikacji uzgadniały numer portu, używany protokołów i sposób formatowania komunikatu.

Drugi podrozdział stanowi istotę tego rozdziału. Przedstawia zarówno teoretyczne, jak i praktyczne podejście do zdalnego korzystania z obiektów. Przykłady kodu ilustrują, jak wybierać i implementować techniki zdalnego wywoływania, aby najlepiej pasowały do danej aplikacji rozproszonej. Podrozdział ten opisuje, jak tworzyć obiekty aktywowane przez klienta i aktywowane przez serwer, jak wybierać sposób formatowania i protokołów, jak instalować zestawy rozproszonych aplikacji i jak używać dzierżaw do zarządzania czasem życia obiektu.

Warto przeczytać ten rozdział przez zapoznanie się z rozdziałem dotyczącym usług Web, który przedstawia drugą technologię platformy .NET do implementacji aplikacji rozproszonych. Choć między zdalnym wywoływaniem a usługami Web występują pewne teoretyczne podobieństwa, technologie te różnią się istotnie ze względu na wydajność, współpracę z innymi technologiami i złożoność implementacji, którą autor aplikacji musi zrozumieć. Lektura tych rozdziałów powinna dać wiedzę wystarczającą do wyboru podjęcia, które najlepiej spełnia wymagania danej aplikacji rozproszonej.

## Domeny aplikacji

Zamstawianie na komputerze platformy .NET oznacza utworzenie wirtualnego, zarządzanego środowiska, w którym można uruchamiać aplikacje. To środowisko izoluje aplikacje przez żądania i kapsułkami używanego systemu operacyjnego. Domeny aplikacji to jedna z kluczowych cech architektury do obsługi zarządzanego środowiska.

Większość systemów operacyjnych postrzega świat w kategoriach procesów, którym przydziela zasoby, na przykład pamięć i tablice potrzebne aplikacjom. Pominięto aplikacje .NET nie mogą działać bezpośrednio w nierządzanych procesach, platforma .NET dzieli proces na logiczne obszary, w których wykonywane są zestawy. Te logiczne obszary to domeny aplikacji. Jak przedstawia to rysunek 14.1, proces może zawierać więcej niż jedną domenę aplikacji, a domena aplikacji może zawierać więcej niż jeden zestaw. Domyślna domena aplikacji powstaje w momencie inicjalizacji wspólnego środowiska uruchomieniowego (CLR), a w razie potrzeby CLR tworzy dodatkowe domeny. Aplikacja może także nakazać CLR utworzenie nowej domeny aplikacji.

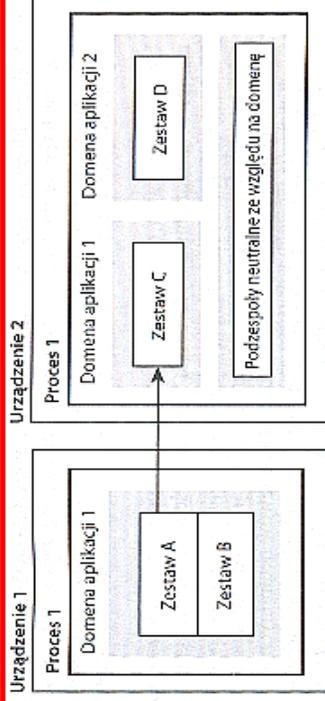
### w domen aplikacji

Oprócz konieczności udostępniania zarządzanego środowiska domeny aplikacji oferują dodatkowe zalety w porównaniu z tradycyjną architekturą opartą na procesach:

- **Izolacja kodu.** Domeny aplikacji stanowią poziom izolacji błędów, dzięki czemu błędy w jednej domenie aplikacji nie powodują problemów w innych domenach. Platforma .NET gwarantuje taką odrębność kodu na dwa sposoby.

### rysunek 14.1.

W pole dynamicznych procesów może znajdować się wiele domen aplikacji



Zapobiega bezpośrednim odwołaniom do obiektów znajdujących się w innych domenach aplikacji, a każda domena aplikacji musi ładować i przechowywać własną kopię kluczowych zestawów, co pozwala na niezależne działanie każdej domeny. Korzystnym produktem ubocznym tego jest możliwość wybiórczego debugowania poszczególnych domen aplikacji i zatrzymywanie ich bez wpływu na działanie innych domen działających w danym procesie.

- **Wydajność.** Implementacja w aplikacji wielu domen może przyczynić się do poprawy wydajności w porównaniu z podobnym projektem opartym na wielu procesach. Lepsza wydajność wynika z kilku czynników: fizyczne procesy wymagają więcej pamięci i zasobów niż domeny aplikacji, które współdziela zasoby jednego procesu. Tworzenie i usuwanie procesów jest dużo bardziej czasochłonne niż podobne operacje w przypadku domen aplikacji. Ponadto przekazywanie wywołań między procesami jest wolniejsze i bardziej kosztowne niż przekazywanie wywołań między domenami aplikacji działającymi w tym samym procesie.

- **Bezpieczeństwo.** Z samej swej natury domeny aplikacji stanowią granice bezpieczeństwa między zawieranymi przez nie zasobami i zestawami próbującymi uzyskać dostęp do tych zasobów. Aby przekroczyć granicę, zewnętrzny zestaw musi polegać na **wywołaniach zdalnych**, co wymaga współpracy między domenami aplikacji. Ponadto domeny aplikacji mają własne strategie bezpieczeństwa, które mogą narzucić zestawom ograniczenie możliwości operacji. Ten ochronny model bezpieczeństwa pozwala domenom aplikacji zagwarantować poprawne zachowanie zestawów. Rozdział 15. opisuje przykłady zastosowań domen aplikacji do zapewnienia bezpieczeństwa kodu.

## Domeny aplikacji i zestawy

Kiedy aplikacja działa, może korzystać z kodu znajdującego się w wielu zestawach. Domyślnie CLR ładuje potrzebne zestawy do domeny aplikacji wywołującego je zestawu. Oznacza to, że jeśli proces zawiera kilka domen aplikacji korzystających z tego samego zestawu, w każdej domenie znajduje się kopia tego zestawu. Jednak możliwa jest zmiana domyślnego działania i współdzielenie jednego zestawu między wieloma domenami. Zestawy używane w ten sposób nazywa się **zestawami neutralnymi ze względu na domenę**.

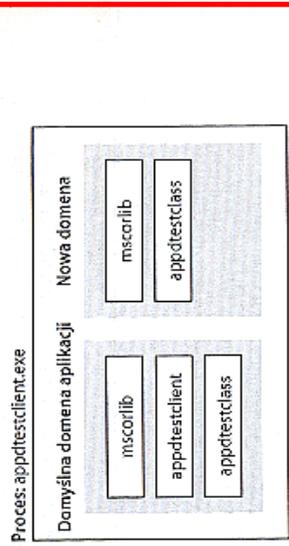
```

{
    Console.WriteLine(
        " Nazwa zestawu: {0}", a.Gatlamia().Name);
}
}
}

```

Rysunek 14.2 przedstawia trzy zestawy znajdujące się w domyślnej domenie aplikacji: mscorlib, appdtestclient i appdtestclass. Platforma .NET automatycznie ładuje zestaw mscorlib, ponieważ zawiera on potrzebną aplikację przestrzeni nazw System.

**Rysunek 14.2.**  
Domyślny zestaw tworzone przez kod Rysunku 14.1



Warto zauważyć, że proces zawiera także drugą domenę, która przechowuje dwa zestawy. Do utworzenia tej domeny służy statyczna metoda CreateDomain():

```
AppDomain myAppDomain = AppDomain.CreateDomain("Nowa domena");
```

W momencie utworzenia obiektu klasy RemotableClass środowisko ładuje dwa zestawy. Jak można tego oczekiwać, wywołanie metody ShowDomain() powoduje wyświetlenie nazwy aktywnej domeny — Nowa domena.

Ostatnia operacja w przykładowym kodzie to usunięcie nowej domeny z procesu. Warto zauważyć, że platforma .NET nie pozwala na usuwanie pojedynczych zestawów z domen aplikacji.

## 2. Zdalne korzystanie z obiektów

Podstawowy cel stosowania technologii Remoting to umożliwienie komunikacji między różnymi domenami aplikacji i wymianę danych między nimi. Zwykle określa się to jako relację klient-serwer, gdzie klient korzysta z zasobów lub obiektów na zdalnym serwerze, kiedy ten pozwoli na taki dostęp. Podstawą zdalnego korzystania z obiektów jest implementacja komunikacji między klientem a serwerem. Fizyczna bliskość domen aplikacji nie ma w tym przypadku znaczenia. Domeny mogą się znajdować w tym samym procesie, w różnych procesach, a nawet na różnych maszynach na różnych kontynentach.

Technologia Remoting często opisuje się jako trudna do zrozumienia i implementacji — szczególnie w porównaniu z usługami Web. W przypadku wielu aplikacji takie wyobrażenie jest mylące i po prostu nieprawdziwe.

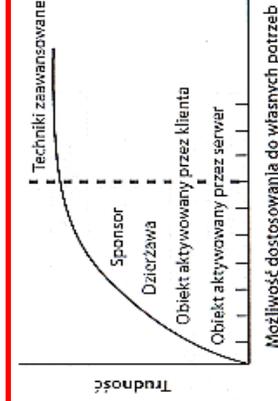
Poniżej wypisane są operacje potrzebne do umożliwienia klientowi dostępu do obiektu na zdalnym serwerze:

- Utworzenie połączenia TCP lub HTTP między klientem a serwerem.
- Wybór sposobu formatowania komunikatów przesyłanych między klientem a serwerem.
- Zarejestrowanie zdalnie używanego typu.
- Utworzenie zdalnego obiektu i aktywacja go przez serwer lub przez klienta.

Za wszelkie szczegóły odpowiada platforma .NET. Nie trzeba rozumieć wszystkich detali dotyczących TCP, HTTP ani portów. Wystarczy określić, że potrzebne jest połączenie, i jaki port ma używać. Jeśli wybrany jest protokół HTTP, do komunikacji wykorzystuje się format SOAP (ang. *Simple Object Access Protocol*). W przypadku protokołu TCP dane przesyłane są w formacie binarnym. Proces rejestrowania zachodzi zarówno po stronie serwera, jak i po stronie klienta. Klient wybiera metodę rejestracji i przekazuje do niej wybranc parametry, które określają adres serwera i używany typ (klasę). Serwer rejestruje typy i porty, które chce udostępnić klientowi, oraz sposób ich udostępniania. Na przykład serwer może udostępnić obiekty typu singleton, tworzone raz i obsługujące wywołania wszystkich klientów. Możliwe jest też tworzenie nowego obiektu do obsługi każdego wywołania.

Rysunek 14.3 przedstawia krzywą uczenia się, której mogą oczekiwać programiści poznający technologię Remoting. Dzięki ukrywanemu większości szczegółów dotyczących komunikacji .NET umożliwia programistom szybkie tworzenie aplikacji korzystających ze zdalnych obiektów. Złożoność często wiązana ze zdalnym korzystaniem z obiektów wchodzi w grę dopiero na wyższym poziomie krzywej i dotyczy wykorzystywania zaawansowanych technik, jak tworzenie dostawców bramek czy niestandardowych kanałów przesyłu. Znajomość tych zaawansowanych technik umożliwia dostosowanie sposobu komunikowania się przez rozproszoną aplikację. Więcej informacji o tych zagadnieniach można znaleźć w książkach na temat zaawansowanych technik zdalnego korzystania z obiektów na platformie .NET, jak *Advanced .NET Remoting* autorstwa Ingo Rammera.

**Rysunek 14.3.**  
Krzywa uczenia się tworzenia aplikacji do obsługi wywołań zwrotnych



Nimniejszy rozdział koncentruje się na zagadnieniach znajdujących się po lewej stronie linii przecinającej krzywą uczenia się. Opisuje, jak projektować aplikacje umożliwiające tworzenie zdalnych obiektów przez serwer lub przez klienta, jak kontrolować czas życia tych obiektów i jak projektować oraz instalować zestawy, aby jak najlepiej wykorzystywać zalety technologii Remoting. W rozdziale znajduje się przykładowy kod przedstawiający schematy.

których można użyć do implementacji rozmaitych aplikacji wykorzystujących zdalne wywołania. Znajdują się wśród nich przykłady serwera przekazującego żądane rysunki klientom oraz serwera przyjmującego wiadomości i przesyłającego je na żądanie do określonych odbiorców.

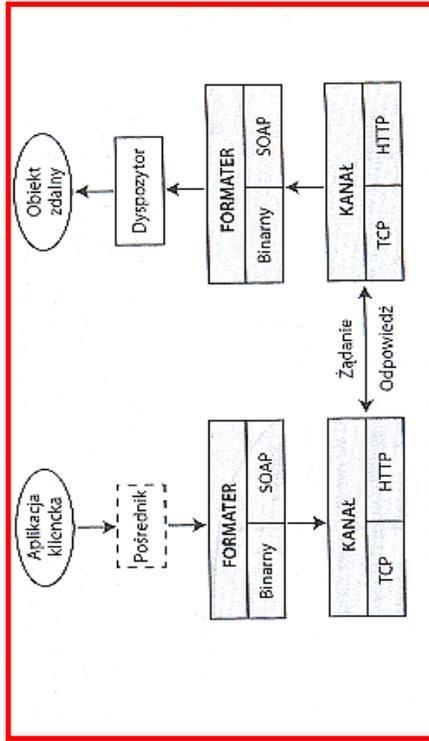
## Architektura technologii Remoting

Kiedy klient próbuje wywołać metodę zdalnego obiektu, wywołanie przechodzi przez kilka warstw na stronie klienta. Pierwsza z nich to pośrednik (ang. *proxy*), którym jest abstrakcyjna klasa o takim samym interfejsie co reprezentowany przez nią zdalny obiekt. Pośrednik sprawdza, czy liczba i typy argumentów w wywołaniu są poprawne, przekształca żądanie na komunikat i przekazuje go do kanału klienckiego. Kanał odpowiada za przekazanie żądania do zdalnego obiektu. W minimalnej wersji na kanał składają się: bramka formatująca (ang. *formatter sink*), która serializuje żądanie na strumień, oraz bramka transportująca (ang. *transport sink*), która przekazuje żądanie do portu na serwerze. Zbiór bramek w kanale to łańcuch bramek. Oprócz tych dwóch standardowych bramek, kanał może zawierać także bramki niestandardowe, które manipulują przesyłanym strumieniem danych.

Po stronie serwera proces ten działa w drugą stronę. Bramka transportująca serwera odbiera komunikat i przekazuje go w górę łańcucha. Po utworzeniu żądania przez bramkę formatującą, platforma .NET tworzy obiekt po stronie serwera i wykonuje żądaną metodę.

Rysunek 14.4 przedstawia rolę klienta i serwera przy obsłudze zdalnych wywołań. W dalszej części opisane są trzy kluczowe komponenty: klasy pośredniczące, formatujące i kanały.

Rysunek 14.4. Ogólny obraz technologii Remoting



## Pośredniki

Kiedy klient próbuje nawiązać komunikację z obiektem zdalnym, referencję do tego obiektu przechowuje tak zwany **pośrednik**. W technologii Remoting są dwa rodzaje pośredników: **pośrednik niewidoczny**, z którym klient komunikuje się bezpośrednio, oraz **pośrednik rzeczywisty**, który pobiera żądanie klienta i przekazuje je do zdalnego obiektu.

Środowisko CLR tworzy niewidoczny pośrednik, aby udostępnić klientowi interfejs identyczny z interfejsem zdalnej klasy. Dzięki temu CLR może sprawdzić, czy wszystkie wywołania klienta pasują do sygnatury docelowej metody (czy odpowiednio są typ i liczba parametrów). Choć CLR odpowiada za tworzenie niewidocznego pośrednika, programista musi zapewnić środowisku dostęp w czasie kompilacji i wykonywania aplikacji do metadanych definiujących zdalną klasę. Najłatwiej to zrobić, przekazując klientowi kopię zestawu serwerów zawierającego klasę. Są jednak lepsze alternatywy, opisane w dalszej części rozdziału.

Kiedy **pośrednik niewidoczny** sprawdzi poprawność wywołania, przekształca je w obiekt komunikatu, którego klasa musi implementować interfejs IMessage. Obiekt komunikatu jest przekazywany jako parametr do metody `Invoke()` rzeczywistego pośrednika, który przekazuje go do kanału. Tutaj obiekt formatujący serializuje komunikat i przekazuje go do obiektu kanału, **który następnie przesyłającego komunikat do zdalnego obiektu**.

Rzeczywisty pośrednik odpowiadający za przesyłanie komunikatu do zdalnego obiektu to implementacja klasy `MarshalProxy` generowana automatycznie przez CLR. Choć spełnia ona wymagania większości aplikacji wykorzystujących wywołania zwrotne, programista ma możliwość utworzenia własnej implementacji.

## Obiekty formatujące

W przestrzeni nazw Remoting platformy .NET znajdują się dwie klasy do obsługi formatowania: **formater binarny i formater SOAP**. Formater SOAP, opisany szczegółowo w rozdziale o usługach Web, serializuje komunikat na format XML. Formater binarny tworzy dużo mniejszy strumień komunikatu niż SOAP, ponieważ przesyła komunikat jako strumień nieprzetworzonych bajtów.

Domyślnie formater SOAP używa się wraz z protokołem HTTP, a formater binarnego z protokołem TCP. Jednak można też przesyłać komunikaty SOAP za pomocą protokołu TCP, a dane binarne za pomocą protokołu HTTP. Choć kombinacja formatu SOAP i protokołu HTTP nie jest tak wygodna jak przesyłanie danych binarnych, stała się standardem przesyłania danych przez firewall zarówno w technologii Remoting, jak i w usługach Web. Format binarny jest zalecany, kiedy firewall nie stanowi problemu.

## Kanały

Obiekty do obsługi kanałów tworzy się na podstawie klas implementujących interfejs `IChannel` Platformy .NET udostępnia dwie klasy, które spełniają większość potrzeb: `HttpChannel` i `TcpChannel`. Za rejestrację kanału służącego do dostępu do zdalnych obiektów odpowiada serwer. Z kolei klient musi zarejestrować kanał, którym chce przesyłać wywołania.

W najprostszej postaci rejestrowanie kanału po stronie serwera polega na utworzeniu obiektu klasy `Channel` i zarejestrowaniu go poprzez przekazanie go jako parametru do statycznej metody `RegisterChannel()` klasy `ChannelServices`. Poniższy kod rejestruje kanał HTTP na porcie 32000:

```
// Rejestracja kanału
HttpChannel c = new HttpChannel(32000);
ChannelServices.RegisterChannel(c); // Port 32000
```

Rejestrowanie kanału po stronie klienta różni się tylko tym, że nie trzeba określać numeru portu:

```
HttpClient c = new HttpClient();
channel = Services.RegisterChannel(c);
```

Przy rejestrowaniu kanałów po stronie klienta i serwera trzeba pamiętać o kilku zasadach:

- Zarówno serwer, jak i klient mogą rejestrować wiele kanałów, jednak klient musi zarejestrować kanał pasujący do kanału zarejestrowanego przez serwer.
- Różne kanały nie mogą używać tego samego portu.
- Domyślne kanały HTTP i TCP mają nazwy http i tcp. Próba zarejestrowania wielu kanałów HTTP lub TCP o domyślnej nazwie spowoduje zgłoszenie wyjątku. Rozwiązaniem jest tworzenie kanałów za pomocą konstruktora przyjmującego jako parameter nazwę kanału. Konstruktor ten opisany jest w dalszej części rozdziału.

Jako alternatywę do umieszczania informacji o kanale i protokole w kodzie .NET można określić je w plikach konfiguracyjnych powiązanych z zestawami klienta i serwera. Na przykład, jeśli nazwa zestawu serwera to *MessageHost.exe*, można utworzyć plik konfiguracyjny o nazwie *MessageHost.exe.config* zawierający poniższą specyfikację portu i formatowania:

```
<application>
  <channels>
    <channel ref="http" port="3200" />
  </channels>
</application>
```

Program używa tego pliku, przekazując jego nazwę do metody `Configure()` klasy `RemotingConfiguration`:

```
RemotingConfiguration.Configure("MessageHost.exe.config");
```

Plik konfiguracyjny musi znajdować się w tym samym katalogu co używany go zestaw.

## Przypisywanie nazwy do kanału

Aby otworzyć wiele kanałów wykorzystujących ten sam protokół, serwer musi przypisać każdemu z nich nazwę. W tym celu trzeba użyć poniższego konstruktora klasy `HttpClientChannel`:

```
HttpClientChannel(channelName,
    IClientChannelSinkProvider csp,
    IServerChannelSinkProvider ssp)
```

Przy nadawaniu nazwy istotny jest tylko pierwszy parametr. Drugiego i trzeciego można użyć do określenia formata używanego po stronie klienta lub serwera:

```
IDictionary chProps = new Hashtable();
chProps["name"] = "HttpClient101";
chProps["port"] = "3202";
ChannelServices.RegisterChannel(new HttpClientChannel(
    null, null));
```

## Rodzaje wywołań zdalnych

Jak opisano to w rozdziałach początkowych, parametry metod języka C# można przekazywać przez wartość lub przez referencję. Wywołania zwrótne używają tych samych technyk do umożliwienia klientom dostępu do obiektów, choć używana tu terminologia jest nieco inna. Sytuację, w której klient pobiera kopię obiektu, nazywa się **szeregowaniem przez wartość**. Jeśli klient pobiera jedynie referencję do zdalnego obiektu, jest to **szeregowanie przez referencję**. Pojęcie szeregowanie oznacza po prostu przekazywanie obiektu lub wywołania między klientem a serwerem.

## Szeregowanie przez wartość

Kiedy obiekt jest szeregowany przez wartość, klient otrzymuje kopię obiektu, która zapisywana jest w domenie aplikacji klienta. Następnie klient może używać tego obiektu lokalnie i nie potrzebuje pośrednika. To rozwiązanie jest dużo mniej popularne od szeregowania przez referencję, gdzie wszystkie wywołania trafiają do zdalnego obiektu. Jednak w przypadku obiektów, które mogą działać po stronie klienta również dobrze co po stronie serwera i są często wywoływane, szeregowanie przez wartość może zmniejszyć koszty przekazywania wywołań do serwera.

Jako przykład posłuży obiekt obliczający indeks masy ciała (ang. *Body Mass Index* - BMI). Zamiast tworzyć obiekt po stronie serwera i zwracać wartości BMI, można zaprojektować rozwiązanie tak, aby zwracać sam obiekt BMI. Klient może używać tego obiektu lokalnie i nie musi przekazywać wywołań do serwera. Poniżej przedstawiony jest sposób implementacji tego rozwiązania.

Aby szeregować obiekt przez wartość, musi być możliwa jego serializacja. Oznacza to, że klasa musi implementować interfejs `ISerializable` lub, co jest rozwiązaniem łatwiejszym, mieć atrybut `[Serializable]`. Poniżej znajduje się kod klasy znajdującej się na serwerze:

```
ISerializable
public class BMICalculator
{
    // Klasa do obliczania indeksu masy ciała
    public decimal inches;
    public decimal pounds;
    public decimal GetBMI()
    {
        return ((pounds * 703 * 10 / (inches * inches)) / 10);
    }
}
```

Klasa `HealthTools` szeregowana przez wartość zwraca obiekt klasy `BMICalculator`:

```
public class HealthTools : MarshalByObject
{
    // Klasa zwraca obiekty do obliczenia BMI
    public BMICalculator GetBMIobj() {
        return new BMICalculator();
    }
}
```

Klient tworzy obiekt klasy HealthTools i wywołuje metodę GetBMI(Obj()), która zwraca obiekt do obliczenia BMI:

```
HealthMonitor monitor = new HealthMonitor();
BMICalculator calc = monitor.GetBMI(Obj());
calc.Pounds = 168M;
calc.Inches = 73M;
Console.WriteLine(calc.GetBMI());
```

Ważne jest, aby zrozumieć, że powyższy przykład wykorzystuje zarówno szeregowanie przez wartość, jak i szeregowanie przez referencję. Typ szeregowany przez referencję (HealthTools) udostępnia metodę GetBMI(Obj()) zwracającą typ szeregowany przez wartość (BMICalculator). Można traktować to jako rodzaj wzorca projektowego fabryki opisanego w rozdziale 4.

### szeregowanie przez referencję

Szeregowanie przez referencję ma miejsce, kiedy klient przekazuje wywołania do obiektu działającego na serwerze zdalnym. Wywołanie jest szeregowane do serwera przez pośrednika, a wyniki wywołania są następnie szeregowane z powrotem do klienta.

Obiekty dostępne poprzez szeregowanie przez referencję muszą dziedziczyć po klasie MarshalByRefObject. Jej najważniejsze składowe, InitializableLifetimeServices i GetLifetimeServices, tworzą i pobierają obiekty służące do kontroli czasu życia zdalnych obiektów na serwerze. Zarządzanie czasem życia obiektu to kluczowa właściwość wywoływania zdalnego, co jest opisane w dalszej części tego punktu.

Sa dwa rodzaje obiektów typu MarshalByRefObject: aktywowane przez klienta i aktywowane przez serwer. Obiekty aktywowane przez serwer dzielą się dalej na obiekty jednomywołaniowe i obiekty typu singleton. Od klienta zależy, który sposób zostanie użyty. Jeśli klient wybierze obiekty aktywowane przez serwer, to serwer określa, czy będzie to obiekt aktywowany na czas jednego wywołania czy typu singleton.

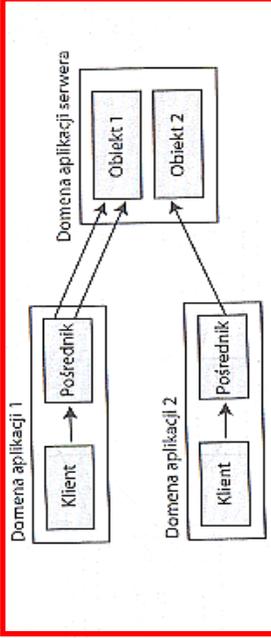
Wybór trybu aktywacji istotnie wpływa na ogólny projekt, wydajność i skalowalność aplikacji obsługującej wywołania zdalne. Wybór ten określa, kiedy obiekty powstają i w jakiej liczbie, jak wygląda zarządzanie ich cyklem życia i czy obiekty przechowują informacje o stanie. Kolejne punkty opisują szczegółowo te zagadnienia.

### obiekty aktywowane przez klienta

Używanie i działanie obiektów aktywowanych przez klienta nie różni się zbytnio od obiektów tworzonych lokalnie. W obu przypadkach obiekty można utworzyć za pomocą operatora new, oba mogą mieć sparametryzowane konstruktory obok konstruktorów domyślnych, a także oba przechowują informacje o stanie we właściwościach lub polach. Jak przedstawia to rysunek 14.5, różnica polega na tym, że obiekty aktywowane przez klienta działają na serwerze w odrębnej domenie aplikacji i są wywoływane przez pośrednik.

rysunek 14.5.

Obiekty aktywowane przez klienta: klient zachowuje kontrolę nad obiektem



Ponieważ obiekt znajduje się w innej domenie aplikacji, podlega procesowi odzyskiwania pamięci w tej domenie i może zostać usunięty, choć klient wciąż chce go używać. Platforma .NET obsługuje ten potencjalny problem, przypisując do każdego obiektu dzierżawę podrzynającą życie obiektów, które mogą jeszcze zostać użyte. Dzierżawy są szczegółowo opisane w dalszej części tego rozdziału.

### obiekty aktywowane przez serwer

Obiekty aktywowane przez serwer można implementować jako obiekty typu singleton lub obiekty jednomywołaniowe. Obiekty typu singleton najlepiej nadają się do współdzielenia pojedynczego zasobu lub wykonywania wspólnych operacji przez wielu użytkowników. Przykładowe aplikacje tego typu to serwer do obsługi rozmów czy fabryka klas. Trybu jednomywołaniowego używa się, kiedy klienci wykonują na serwerze względnie krótkie operacje. Klienci wymagają przechowywania informacji o stanie między poszczególnymi wywołaniami. To podejście najlepiej się skaluje i ma zaletę w postaci normalnego funkcjonowania w środowiskach wymagających równoważenia obciążenia bezpośrednich wywołań do wielu serwerów.

### Obiekty typu singleton

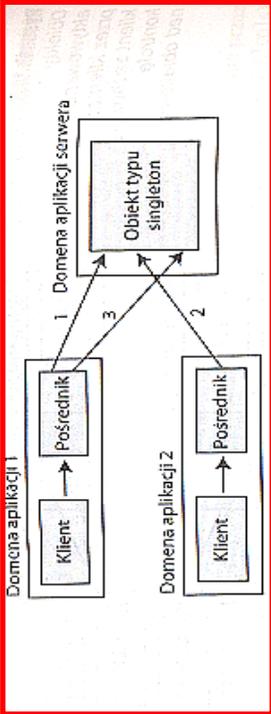
Rysunek 14.6 przedstawia, w jaki sposób pojedynczy obiekt obsługuje wszystkie wywołania w projekcie typu singleton. Serwer tworzy obiekt, kiedy pierwszy klient próbuje uzyskać do niego dostęp, a nie w momencie próby utworzenia go. Poniżej serwer tworzy obiekt tylko raz, próby utworzenia obiektu przez innych klientów są ignorowane. W zamian wszyscy klienci otrzymują referencję do tego samego obiektu typu singleton. Przy każdym wywołaniu obiektu przez klienta środowisko CLR przydziela nowy wątek z puli wątków. Z tego powodu programista powinien zapewnić bezpieczeństwo kodu serwera ze względu na wątki. Ta technika ogranicza także skalowalność, ponieważ zwykle dostępne jest tylko ograniczona liczba wątków.

### Obiekty jednomywołaniowe

W trybie jednomywołaniowym serwer tworzy nowy obiekt za każdym razem, kiedy klient wywołuje obiekt. Po obsłużeniu tego wywołania ma miejsce deaktywacja obiektu. Rysunek 14.7 ilustruje obsługę wielu wywołań. Pierwszy wywołanie zostało obsłużone, a utworzon

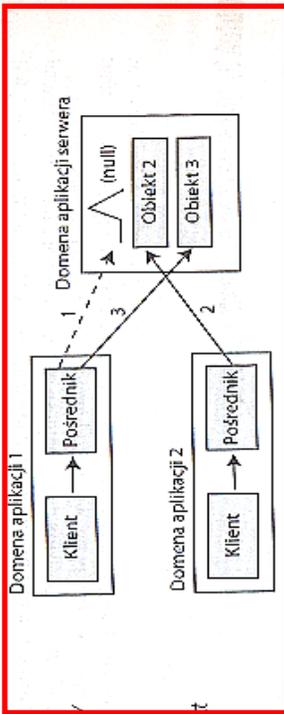
rysunek 14.6.

obiekt typu singleton  
wywołany przez serwer  
- jeden obiekt  
obsługuje wszystkie wywołania



rysunek 14.7.

obiekt jednowywołaniowy  
wywołany przez serwer  
- przy każdym zadaniu serwer tworzy nowy obiekt



do tego obiekt jest usunięty. Na tym etapie pośrednik zgłaszającego to wywołanie przechowuje referencję na obiekt null. Drugie wywołanie pochodzi od innego klienta i powoduje utworzenie drugiego obiektu na serwerze. W końcu pierwszy klient zgłasza drugie wywołanie, a serwer tworzy trzeci obiekt.

Zaletą aktywacji jednowywołaniowej jest zwalnianie zasobów natychmiast po obsłużeniu wywołania. Kontrastuje to z wywołaniami aktywowanymi przez klienta, gdzie klient przechowuje zasoby do czasu zakończenia używania obiektu. Wadą obiektów jednowywołaniowych jest to, że nie udostępniają naturalnego sposobu przechowywania informacji o stanie między kolejnymi wywołaniami. Jeśli potrzebna jest skalowalność wynikająca z używania obiektów jednowywołaniowych, a jednocześnie ważne są informacje o poprzednich wywołaniach, trzeba zaprojektować serwer tak, aby przechowywał informacje o stanie w pliku lub w bazie danych.

## Rejestracja typów

Aplikacja może obsługiwać wiele trybów aktywacji i wiele obiektów. Klient wskazuje obiekty na serwerze, których chce używać, oraz tryb ich aktywacji. Z kolei serwer określa, które obiekty udostępnią zdalnym klientom oraz tryb aktywacji potrzebny w celu dostępu do nich. Służy do tego mechanizm zwany **rejestracją typów**. Jako uzupełnienie rejestracji kanałów, która informuje .NET o sposobie przekazywania komunikatów, rejestracja typów określa dostępne zdalne obiekty oraz tryb ich aktywacji. Jest to ostatni element uzgadniania między aplikacjami. Klienci w jednej domenie aplikacji uzyskają dostęp do obiektów aktywowanych w innej domenie aplikacji.

## Rejestrowanie obiektów aktywowanych przez serwer

Zestaw serwera używa do rejestracji typu metody RegisterWellKnownServiceType() klasy RemoteConfiguration. Ta metoda przyjmuje trzy parametry: typ obiektu, ciąg znaków reprezentujący URI obiektu (uniwersalny identyfikator zasobu, ang. *universal resource identifier*) i wartość wylczenia WellKnownObjectType, która określa, czy obiekt ma być jednowywołaniowy czy typu singleton. Poniższy fragment kodu rejestruje obiekt MessageHandler tworzący jako obiekt typu singleton.

```

// Rejestracja po stronie serwera — obiekty aktywowane przez serwer
Type ServerType = typeof(SimpleServer.MessageHandler);
RemoteConfiguration.RegisterWellKnownServiceType(
    ServerType, // Typ obiektu
    "MsgObiekt", // Długość nazwa
    WellKnownObjectType.Singleton);
  
```

Zamiast wartości Singleton na `WellKnownObjectType` powoduje zarejestrowanie obiektu w trybie jednowywołaniowym.

W celu dostępu do obiektów aktywowanych przez serwer klient używa metody RegisterWellKnownClientType(). Ta metoda przyjmuje dwa parametry: typ obiektu oraz ciąg znaków zawierający adres URL lokalizacji obiektu. Warto zauważyć, że klient nie ma wpływu na to, czy zdalny obiekt jest typu singleton czy jest jednowywołaniowy. Musi używać tego, który udostępnił serwer.

```

// Rejestracja po stronie klienta — obiekty aktywowane przez serwer
Type ServerType = typeof(SimpleServer.MessageHandler);
string url = "http://locaHost:3200/MsgObiekt";
// Rejestracja typ w trybie aktywowany przez serwer
RemoteConfiguration.RegisterWellKnownClientType(
    ServerType,
    url);
MessageHandler m = new MessageHandler();
  
```

Kiedy klient używa instrukcji new do tworzenia instancji zdalnego obiektu, .NET rozpoznaje, że dany obiekt jest zarejestrowany i używa adresu URL do jego zlokalizowania.

## Rejestrowanie obiektów aktywowanych przez klienta

Serwer do rejestracji obiektów aktywowanych przez klienta używa metody RegisterActiveTypedServiceType(). Ta metoda wymaga tylko jednego parametru — typu obiektu:

```

// Rejestracja po stronie serwera — obiekty aktywowane przez klienta
Type ServerType = typeof(ImageServer); // Klasa ImageServer
RemoteConfiguration.RegisterActiveServiceType(
    ServerType);
  
```

Rejestracja po stronie klienta jest niemal równie prosta. Wymaga wywołania metody RegisterActiveClientType() i przekazania do niej typu obiektu oraz adresu URL określającego lokalizację obiektu:

## Zdalne wywołania obiektów aktywowanych przez serwer

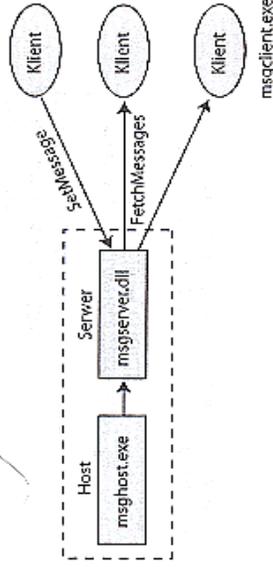
Wiedząc, jak rejestrować kanały i typy, można przystąpić do tworzenia aplikacji wykorzystującej zdalne wywołania. Pierwszy przykład to aplikacja umożliwiająca użytkownikom przesłanie wiadomości do innych użytkowników i ich pobieranie. Rozwiązanie oparte jest na obiektach aktywowanych przez serwer. Poniżej przedstawione są dwa sposoby utworzenia takiej aplikacji. Następnym przykładem używa obiektów aktywowanych przez klienta do pobierania rysunków z serwera.

## serwer do przekazywania wiadomości

Minimalne wymagania aplikacji używającej zdalnych wywołań to zestaw zawierający kod kliencki oraz zestaw działający jako serwer i udostępniający kod zdalnych obiektów. Jak pokazuje to rysunek 14.8, najbardziej popularny model wymaga trzech zestawów: klienta, serwera obsługującego rejestrację kanałów i typów oraz serwera zawiązującego kod obiektów. Właśnie ten model posłuży do utworzenia pierwszego projektu serwera do przekazywania komunikatów.

### Rysunek 14.8.

W przykładzie do zdalnego serwera do przekazywania wiadomości użyte są trzy zestawy



Przed analizą kodu trzeba ustalić słownictwo używane do opisu zestawów. W niniejszym rozdziale serwer oznacza zestaw z deklaracją i implementacją zdalnych klas, a *host* to zestaw zawierający kod do rejestracji typów i kanałów. Jeśli funkcje hosta i serwera są połączone, taki zestaw to serwer lub serwer-host. W literaturze opisującej zdalne wywołania niektórzy autorzy odwracają znaczenie hosta i serwera, podczas gdy inni opisują host jako *ogólny* zestaw.

Aplikacja do przekazywania komunikatów składa się z trzech plików źródłowych, kompilowanych do zestawów *msgserver.dll*, *msghost.exe* i *msgclient.exe*:

```
gsc /t:library msgserver.cs
gsc /r:msgserver.dll msghost.cs
gsc /r:msgserver.dll msgclient.cs
```

Warto zauważyć, że kod serwera jest kompilowany do biblioteki (DLL), a host i klient muszą mieć do niej referencje w czasie kompilacji.

## Zestaw serwera

Listing 14.2 zawiera kod klasy `MessageServer`, która jest udostępniana klientom jako obiekt typu singleton aktywowany przez serwer. Oprócz wymaganego dziedziczenia po klasie `MarshalByRefObject` klasa ta jest miodródmialna od zwykłych klas, nieprzeznaczonych do

```
// Rejestracja po stronie klienta — obiekty aktywowane przez klienta
Type ServerType = Type.FromName("ImageServer");
RemoteRegistration.RegisterActivatedClientType(
    ServerType,
    "tcp://localhost:3201");
```

## Rejestracja typów w pliku konfiguracyjnym

Podobnie jak w przypadku kanałów instrukcje służące do rejestracji typów można umieścić w pliku konfiguracyjnym zestawu. Dwa poniższe fragmenty kodu ilustrują, jak zarejestrować po stronie serwera i po stronie klienta aktywowany przez serwer obiekt z poprzedniego przykładu:

```
// Plik serwera — rejestracja obiektu MessageManager w typie singleton
<application>
  <services>
    <wellKnown
      mode="Singleton"
      type="SimpleServer.MessageManager, msgserver"
      objectUrl="MojoObiekt" />
    </services>
  </application>
```

```
// Plik klienta — rejestracja obiektu MessageManager na porcie 3200
<application>
  <client>
    <wellKnown
      type="SimpleServer.MessageManager, msgserver"
      url="http://localhost:3200/MojoObiekt" />
    </client>
  </application>
```

Warto zauważyć, że informacje potrzebne do rejestracji są przedstawione jako atrybuty znacznika `<wellKnown>`, a atrybut `type` opisuje obiekt za pomocą przestrzeni nazw i nazwy, jak również zawierającego go zestawu.

Rejestracja obiektów aktywowanych przez klienta wymaga użycia znacznika `<activated>` do określenia zdalnego obiektu w plikach konfiguracyjnych serwera i klienta.

```
// Plik serwera — rejestracja obiektu ImageServer jako aktywowanego przez klienta
<application>
  <services>
    <activated type="ImageServer, caoimageserver" />
  </services>
</application>
```

Plik klienta zawiera ponadto atrybut `url` udostępniający adres zdalnego obiektu:

```
// Plik klienta — rejestracja obiektu ImageServer jako aktywowanego przez klienta
<application>
  <client url="tcp://localhost:3201">
    <activated type="ImageServer, caoimageserver" />
  </client>
</application>
```