

.NET Remoting

Gestão do tempo de “vida” dos objectos remotos

Gestão do tempo de “vida” dos objectos remotos

Num ambiente distribuído o conceito de *Automatic Garbage Collection*, é complexo, pois não existe forma de o *runtime* de uma máquina controlar as referências, para os objectos, ainda válidas noutras máquinas. Por exemplo, um cliente numa máquina pode obter uma referência para um objecto remoto e depois terminar, eventualmente por erro, sem qualquer notificação ao servidor que já não necessita da referência do objecto.

- Em DCOM e CORBA é usado um conceito idêntico ao existente em COM.
- Em COM, existe um contador de referências controlado por métodos que todos os objectos têm de implementar, *AddRef()* e *Release()*, sendo responsabilidade do programador garantir a consistência do número de referências e destruir o objecto quando o contador fica a zero.
- Em DCOM o cliente faz *ping* ao servidor a certos intervalos de tempo para indicar que ainda está “vivo”.

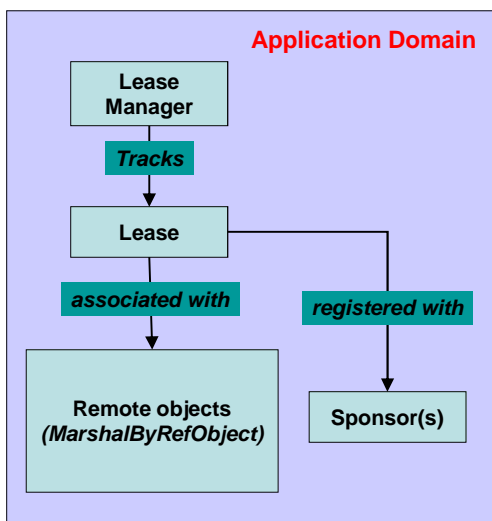
Este modelo introduz *overhead* devido à invocação de métodos remotos, *AddRef()* e *Release()*, sem benefício real para a aplicação.

Lease-based lifetime

O Java RMI, introduziu o conceito de serviço *Lease-based lifetime* (arrendamento de tempo de vida), sendo seguido um conceito idêntico em .NET:

- A cada objecto remoto (*MarshalByRefObject*) criado, é atribuído um tempo de vida – *time-to-live* (TTL) suportado por um objecto *Lease*;
- Existe no *runtime* um gestor de leases (*LeaseManager*) que faz, periodicamente, *polling* a todos os objectos *Lease* associados aos objectos remotos (*MarshalByRefObject*) sendo decrementado o valor de TTL;
- Quando o valor de TTL fica a zero, o objecto é marcado como *timed out*, ficando assim elegível para *Garbage Collection*;
- Cada vez que é feita uma chamada a um método no objecto é actualizado o tempo de TTL (*Renew On Call*) por forma a adiar o *timed out*;
- É ainda possível associar ao *Lease* do objecto remoto um objecto *sponsor* que quando o TTL fica a zero é interrogado pelo *runtime* para responder se o TTL do objecto remoto é ou não renovado. Os *sponsors* são também objectos *MarshalByRefObject* e podem ficar do lado do servidor, do lado do cliente ou noutra máquina a que o *runtime* tenha acesso.

Lease-based lifetime



- Cada *AppDomain* contém um *Lease Manager* que mantém referências para os objectos *lease* dos objectos *MarshalByRefObject* instanciados nesse *AppDomain* e que tenham sido expostos para fora desse *AppDomain*.
- Estão neste caso os objectos (*Singleton*, *Client Activated*, ou objectos expostos por publicação (*Marshal*) ou pelo padrão *Factory*.
- Cada *lease* pode ter associado **zero ou mais *sponsors*** que estão encarregues de renovar ou não o tempo de *lease* quando o *Lease Manager* determina que o tempo de *lease* expirou.

Objectos Lease

- Quando o CLR verifica que um objecto *MarshalByRefObject* fica acessível remotamente, isto é, existem referências fora do contexto do objecto, solicita ao objecto um *Lease* com o tempo de vida, invocando o método *InitializeLifetimeServices*, que é herdado da classe *MarshalByRefObject*;
- Um *lease* é um objecto que encapsula valores *TimeSpan* usados para gerir o *time-to-live (TTL)* dos objectos remotos;
- Existe uma interface (*ILease*) para modelar esta funcionalidade;

Redefinição do tempo de vida (TTL) de um objecto

- O método *InitializeLifetimeServices* pode ser *override* para devolver um *Lease* com propriedades diferentes das existentes por omissão;

```
class SomeMBRType : MarshalByRefObject {
    ...
    public override object InitializeLifetimeService( ) {
        // Retornar o objecto Lease a null, significa que o TTL nunca expira,
        // isto é, o objecto passa a ter tempo de vida infinito
        return null;
    }
    ...
    public void someMethod() {
        ...
        ILease lease =(ILease)this.GetLifetimeService();
    }
}
```

Interface *ILease* e classe *Lease*

namespace: `System.Runtime.Remoting.Lifetime`

Properties class <i>Lease</i>	Time default	Descrição
<i>InitialLeaseTime</i>	5 minutos	Tempo de vida (TTL) inicial quando um objecto é criado.
<i>RenewOnCallTime</i>	2 minutos	Tempo de renovação quando um método do objecto é invocado. Quando um método é chamado o CLR determina o tempo que resta até o <i>Lease</i> expirar. Se o tempo que resta for menor que o indicado em <i>RenewOnCallTime</i> o TTL é renovado para o valor indicado em <i>RenewOnCallTime</i> . Por exemplo, em sucessivas chamadas a um método temos como TTL corrente (assumindo tempos por omissão): 5 – 4.3 – 4 – 3 – 2.5 – 2 – 2 – 2 ...
<i>SponsorShipTimeout</i>	2 minutos	Tempo de espera pela resposta de um possível <i>sponsor</i> . Quando um <i>Lease</i> tem associado um <i>sponsor</i> , este é contactado para que possa renovar junto do <i>LeaseManager</i> o tempo de vida para o objecto.

InitializeLifetimeService – *LeaseState* values

When overriding [InitializeLifetimeService](#) you must check the value of the [CurrentState](#). You can only change the lease values when [CurrentState](#) equals Initial. The only call that affects the lifetime service is the call to [InitializeLifetimeService](#) from the .NET remoting infrastructure, which activates the lease. Other code can call [InitializeLifetimeService](#) and create a lease, but that lease stays in its initial state until it is returned to the .NET remoting infrastructure. If [InitializeLifetimeService](#) returns null, the object's lifetime is infinite and is not garbage collected until the hosting application domain is unloaded.

The implementation of [InitializeLifetimeService](#) normally calls the corresponding method of the base class to retrieve the existing lease for the remote object. If the object has never been marshaled, the lease returned is in its initial state and the lease properties can be set. Once the object has been marshaled, the lease goes from the initial to the active state and any attempt to initialize the lease properties is ignored and an exception is thrown. [InitializeLifetimeService](#) is called when the remote object is activated. A list of sponsors for the lease can be supplied with the activation call and additional sponsors can be added at any time while the lease is active. ([http://msdn.microsoft.com/en-us/library/9f82k54b\(v=VS.100\).aspx](http://msdn.microsoft.com/en-us/library/9f82k54b(v=VS.100).aspx))

LeaseState values	Description
Null	The lease is not initialized.
Initial	The lease has been created, but is not yet active.
Active	The lease is active and has not expired.
Renewing	The lease has expired and is seeking sponsorship.
Expired	The lease has expired and cannot be renewed.

Objecto que inicia o Serviço de *Lifetime*

```
class Aluno : MarshalByRefObject, IRemAluno {  
  
    public override object InitializeLifetimeService() {  
        ILease lease = (ILease)base.InitializeLifetimeService(); // obtém o Lease da classe base  
        if (lease.CurrentState == LeaseState.Initial) {  
            lease.InitialLeaseTime=TimeSpan.FromSeconds(20);  
            lease.RenewOnCallTime=TimeSpan.FromSeconds(5);  
            // lease.SponsorshipTimeout=TimeSpan.FromSeconds(10);  
            lease.SponsorshipTimeout=TimeSpan.Zero; // Lease sem sponsor  
        }  
        return lease;  
    }  
    // ... Implementação da classe  
    // Acesso num método ao tempo de Lease corrente  
    public string alunoHello() {  
        Console.WriteLine("execução do método AlunoHello() {0}",myNome);  
        ILease lease =(ILease)this.GetLifetimeService(); // Obtem o Lease corrente  
        Console.WriteLine("Tempo de Lease corrente {0}\n", lease.CurrentLeaseTime);  
        return "Hello " + myNome;  
    }  
}
```

TimeSpan tem métodos para tempo em milissegundos, segundos, minutos, horas, dias, Zero, MaxValue, MinValue.

Em qualquer momento uma instância pode saber o seu TTL corrente, excepto se Lease==null

Tratamento de excepções no Cliente por termino do tempo de vida

CLIENTE:

Chamada do método *alunoHello()* de um objecto do tipo *aluno* com tratamento da excepção por fim do tempo de vida

```
try {  
    for (int i=0;i<10;i++) {  
        Console.WriteLine("{0}\n", jose.alunoHello());  
        Thread.Sleep((i+1)*1000);  
    }  
} catch (Exception e) {  
    Console.WriteLine("Fim de vida do objecto Aluno:{0}",e.Message);  
}
```

O serviço de *Lifetime* numa aplicação pode ser usado, por exemplo, para alterar o tempo de *polling* do *LeaseManager* que por omissão é de 10 segundos.

LifetimeServices.LeaseManagerPollTime=TimeSpan.FromSeconds(1);

LifeTime Sponsors

Objecto *MarshalByRefObject* que implementa a interface *ISponsor*

```
public interface ISponsor {  
    TimeSpan Renewal(ILease lease);  
}
```

```
public class MySponsor: MarshalByRefObject, ISponsor {  
    public override object InitializeLifetimeService() {  
        return null; // O Sponsor fica com tempo de vida infinito  
    }  
  
    public bool IsForRenew = true;  
  
    public TimeSpan Renewal(ILease lease) {  
        Console.WriteLine("{0}: Sponsor chamado", DateTime.Now);  
        if (IsForRenew) {  
            Console.WriteLine("{0}: vai renovar 10 seg.", DateTime.Now);  
            return TimeSpan.FromSeconds(10);  
        } else {  
            Console.WriteLine("{0}: Não renova mais.", DateTime.Now);  
            return TimeSpan.Zero;  
        }  
    }  
} // end class MySponsor
```

O método *Renewal* vai ser chamado em *callback*, pelo que a classe deve ser *MarshalByRefObject* e o canal tem de ser *Full Serializable*

Registo do *Sponsor* do lado do cliente

```
...  
// criar o objecto remoto  
IRemAluno jose = fact.getNewInstanceAluno("jose");  
// Criar um sponsor para o objecto remoto  
MySponsor sponsorjose = new MySponsor();  
// Obter o Lease do objecto.  
ILease leasejose =  
    (ILease)RemotingServices.GetLifetimeService((MarshalByRefObject)jose);  
  
// Registrar o sponsor no LifetimeService do objecto remoto  
leasejose.Register(sponsorjose);  
...
```

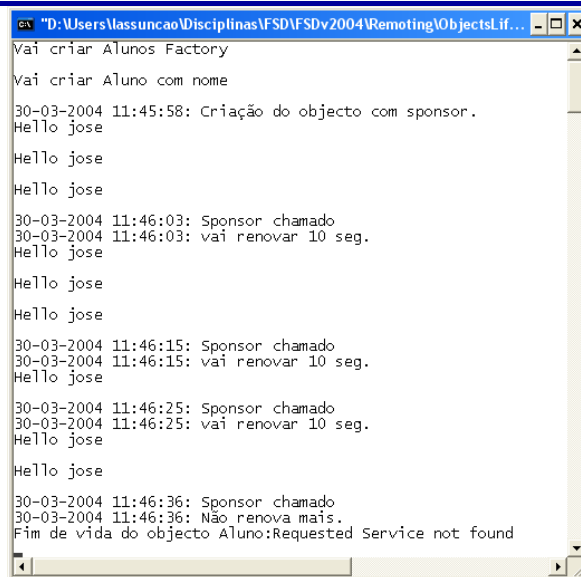
Quando o cliente quiser remover o *sponsor* faz:

```
leasejose.Unregister(sponsorjose);
```

Código do cliente para demonstrar o funcionamento do *sponsor*

```
...
try {
    for (int i=0;i<10;i++) {
        Console.WriteLine("{0}\n",jose.alunoHello());
        //Console.ReadLine();
        if (i == 7) sponsorjose.IsForRenew=false;
        Thread.Sleep((i+1)*1000);
    }
} catch (Exception e) {
    Console.WriteLine("Fim de vida do objecto Aluno:{0}",e.Message);
}
...
```

Resultado de execução do cliente



```

D:\Users\lasmuncao\Disciplinas\FSD\FSDv2004\Remoting\Objects\If...
Vai criar Alunos Factory
Vai criar Aluno com nome
30-03-2004 11:45:58: Criação do objecto com sponsor.
Hello jose
Hello jose
Hello jose
30-03-2004 11:46:03: Sponsor chamado
30-03-2004 11:46:03: vai renovar 10 seg.
Hello jose
Hello jose
Hello jose
30-03-2004 11:46:15: Sponsor chamado
30-03-2004 11:46:15: vai renovar 10 seg.
Hello jose
30-03-2004 11:46:25: Sponsor chamado
30-03-2004 11:46:25: vai renovar 10 seg.
Hello jose
Hello jose
30-03-2004 11:46:36: Sponsor chamado
30-03-2004 11:46:36: Não renova mais.
Fim de vida do objecto Aluno:Requested Service not found
```

Sponsor no lado do servidor

Quando se usa o *sponsor* do lado do cliente é necessário garantir que o cliente é acedido pelo servidor. Quando o cliente está por detrás de um *firewall* ou um *web proxy* os *sponsors* terão de estar do lado do servidor.

O exemplo apresentado nas aulas tem a possibilidade da aplicação cliente criar o *sponsor* no lado do servidor:

(Estudar o exemplo: *ObjectsLifetime.zip*)

Mas se o cliente falhar antes de realizar a operação de *Unregister* do *sponsor* acontece poderem ficar objectos indefinidamente no servidor a ocupar recursos?

Exercício: Proponha uma solução para este problema.