**Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software**

# Multicamadas

## Ótimas razões para migrar sua solução client/server

***Resumo DevMan***

O desenvolvimento com o DataSnap no Delphi passou por várias fases. Entender como funciona o DataSnap internamente, suas vantagens e recursos, é essencial para a construção de aplicações corporativas, sejam elas multicamadas ou client/server. O DataSnap permite a construção de soluções mais robustas, com maior escalabilidade e performance, aliado aos poderosos recursos do COM+.

***Nesse artigo veremos***

- Arquitetura de aplicações multicamadas;
- Vantagens do desenvolvimento DataSnap e COM+;
- Funcionamento do dbExpress e DataSnap;
- Aplicações client/server com DataSnap local;
- Aplicações multicamadas com recursos avançados do COM+, como Object Pooling;

***Qual a finalidade***

- Mostrar como construir aplicações multicamadas baseadas em aplicações client/server.

***Quais situações utilizam esses recursos?***

- Mesmo aplicações client/server podem fazer bom uso de técnicas multicamadas. Uma aplicação dbExpress bem escrita é uma aplicação DataSnap local. Qualquer aplicação, seja ela planejada para uma solução corporativa ou não, pode fazer uso das técnicas aqui apresentadas.

**Guinther Pauli**

*(guinther@devmedia.com.br)*

é Delphi Certified: 6, 7, 2005, 2006, Web e Kylix e Microsoft Certfied: MCP, MCAD, MCSD.NET, MCTS e MCPD. Desenvolve aplicações multicamadas desde o MIDAS no Delphi 3/4. Já trabalhou em grandes projetos multicamadas, com mais de 3000 units clientes e 500 objetos servidores COM+.


O desenvolvimento multicamadas com o Delphi é algo encantador. Além de todas as facilidades que temos ao usar esse tipo de arquitetura, que cito a seguir, temos a facilidade do ambiente de desenvolvimento RAD do Delphi. Confesso que não vi até hoje um framework tão produtivo quanto o *DataSnap* para desenvolver soluções para tal propósito.

O objetivo deste artigo é apresentar técnicas intermediárias e avançadas de desenvolvimento com o *DataSnap* do Delphi, para que você definitivamente migre sua solução para multicamadas, mesmo que a solução vá rodar em uma única máquina. Apresentarei duas técnicas para este propósito, a de simular um ambiente *multi-tier* em *client/server* e como rodar o servidor como uma *library*.



Clube101.indb 26 10.11.08 15:52:56

Se o leitor lembrar, na edição 97, vimos como construir nossas primeiras aplicações multicamadas com o Delphi, fazendo chamadas remotas de procedimento usando COM+, sem, no entanto usar banco de dados. Neste artigo vou demonstrar técnicas de desenvolvimento *DataSnap*, ao mesmo tempo que apresento boas práticas de programação para essa arquitetura, usando agora um banco de dados. Partiremos do pressuposto que o leitor tenha uma aplicação *client/server* tradicional, com uma aplicação cliente *Win32* acessando um servidor *Firebird*.

## Protocolos

Quando falamos em aplicações multicamadas, estamos falando em diferentes computadores se comunicando. Isso exige um protocolo. O primeiro passo é escolher qual o melhor protocolo para ser utilizado na comunicação. Atualmente, os protocolos suportados pelo *DataSnap* são:

*DCOM* – Você usa DCOM ao utilizar *Remote Data Modules*, ou *Data Modules* remotos. Não é uma boa solução pois não é escalável, visto que seus objetos são *state-full*, ou seja, para cada cliente há um objeto ativo no servidor, respondendo às requisições;

*COM+* - Originado do *MTS*, o COM+ é a melhor solução para ambientes corporativos (intranet), pois é mais escalável, visto que seus objetos são *state-less*, ou seja, cada objeto é destruído no servidor após processar a requisição cliente;

*SOAP* – O suporte ao *Simple Object Access Protocol* surgiu no *Delphi 6*, e é recomendado quando sua solução precisa ter clientes espalhados pela Internet. É mais lento que o COM+ para ambientes corporativos, justamente por utilizar XML para intercâmbio de dados. Para mais informações sobre *DataSnap* com SOAP, recomendo a leitura do meu artigo na edição 70;

*WebConnection* – descontinuado pois desempenha basicamente a mesma função do SOAP;

*CORBA* – formato proprietário que teve a intenção de ser multi-plataforma. Descontinuado, recomendo a utilização do SOAP para estes casos;

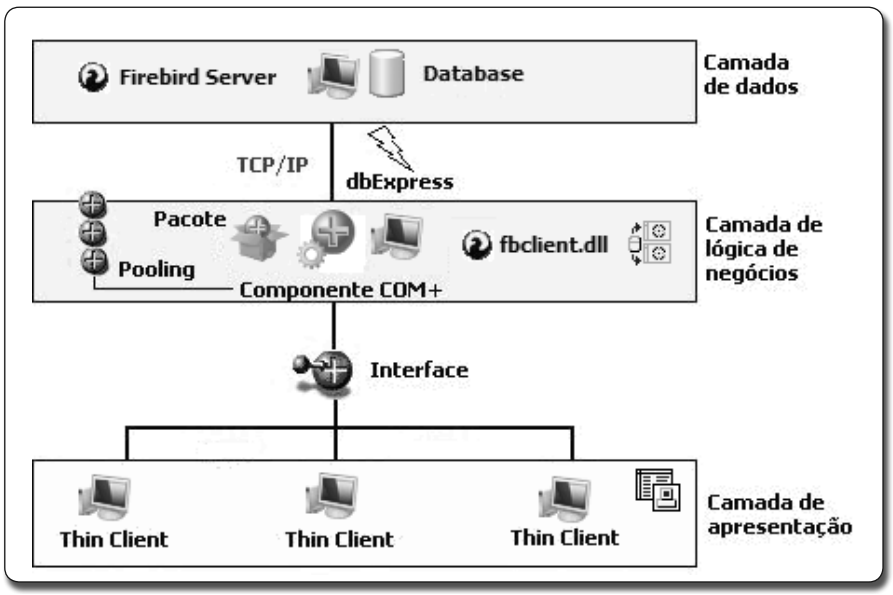



**Figura 1**. Solução multicamadas

*BSS – Borland Socket Server*, baseado no DCOM, também não recomendado para ambientes corporativos por ser *state-full*. Substituído pelo *TDsDataModule* no *Delphi 2009*;

Em nossa solução, vamos optar pelo melhor e mais óbvio, o COM+.

## Arquitetura

Nada melhor que uma ilustração para representar como é formada uma solução *DataSnap*. Na **Figura 1,** temos um diagrama da solução que será construída. Ele servirá como nosso "mapa", sempre que tiver alguma dúvida sobre qual camada está sendo empregado determinado código, consulte a figura.

Em uma arquitetura multicamadas, temos vários computadores que desempenham papéis diferentes na solução. É óbvio que você pode usar um único computador para criar a aplicação *DataSnap*, porém as camadas físicas ainda existirão. Cada processo que roda em diferentes máquinas é também conhecido como *camada*. Em nossa figura, podemos identificar 3 camadas:

*Camada de Dados* – é o banco de dados em si, nesse caso, estamos utilizando o *Firebird*. Para acesso ao banco, o *engine* utilizado aqui é o *dbExpress* (nada impede que se utilize outro, como *ADO*, *IBX*, *Zeos*, *BDE* etc., desde que o *DataSet* suporte a interface *IProviderSupport*, discutida a seguir). Uma outra boa solução que recomendo, principalmente para novos projetos, é utilizar o *SQL Server* com *ADO*, pois se integra melhor ao COM+;

*Camada Lógica de Negócios* – também conhecido como *Servidor de Aplicação* (ou simplesmente *AppServer* para os mais íntimos), responsável por concentrar o acesso ao banco de dados e hospedar os componentes COM+. As aplicações cliente NÃO enxergam diretamente o BD, apenas se comunicam com o servidor de aplicação através de uma interface COM. É nessa camada que você centralizará as regras de negócio da aplicação, como por exemplo, validar uma compra, verificar a disponibilidade de um produto em estoque (consultando o BD), calcular algum imposto, realizar alguma validação etc. É nessa camada que residem os componentes do *dbExpress* e *DataSetProviders*;

*Camada de Apresentação* – é nessa camada que está a menor parte do código. Digo menor porque você basicamente criará telas para exibir os dados oriundos do servidor de aplicação e *database*. Essa camada não vê o BD, não tem *dbExpress* nem qualquer acesso direto ao servidor *Firebird*. Tudo é feito através da chamada de métodos COM ao servidor de aplicação. Por esse motivo, aplicações clientes em uma arquitetura *DataSnap* são também conhecidas como *Thin-Clients* (clientes leves ou magros);



Clube101.indb 27 10.11.08 15:52:58

**Listagem 1.** IAppServer

```
IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;

MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer;
      out RecsOut: Integer; Options: Integer; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString; Data: OleVariant):
      OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString; var OwnerData: OleVariant):
      OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
      RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant); safecall;
end;
```

Como comentei, a comunicação entre as camadas exige um protocolo. Do servidor de aplicação para o BD, o protocolo utilizado é o TCP/IP, normal de uma comunicação *client/server* com o *Firebird*. O tipo de BD independe da solução, você pode usar *SQL Server*, *DB2*, *Oracle* etc.

A comunicação do *thin-client* com o servidor de aplicação é mais interessante. Ela usa um protocolo chamado COM (*Component Object Model*), através de uma interface. Imagine uma interface COM como um controle remoto de um DVD Player. Você apenas aciona comandos através dele, mas todo o processamento ocorre remotamente. O cliente precisa apenas conhecer a interface do servidor para se comunicar com ele, não toma conhecimento sobre como funcionam seus processos internos, como cálculos, validações e processamento de regras de validação, muito menos sobre o BD que está utilizando e como é feita a comunicação. No *DataSnap*, a interface base para a comunicação se chama *IAppServer*.

## IAppServer

A interface *IAppServer* surgiu no Delphi 5, para substituir as então interfaces *IProvider* e *IDataBroker*. O motivo da substituição foi a inclusão do suporte a servidores *state-less*, como o caso do MTS/COM+. Além disso, houve uma falha de arquitetura ao se planejar o antigo *MIDAS* (versões 1 e 2). Para quem lembrar, cada *DataSetProvider* utilizado em um *Data Module* remoto exigia a criação de um novo método COM, o que obrigava a modificação da interface e a necessidade de um novo registro.

*IAppServer* resolve esse problema elegantemente, com uma solução simples: o nome do *DataSetProvider* é passado como parâmetro nas chamadas dos métodos COM, o que não exige um novo método para cada tabela que você queira acessar no banco. Além disso, a modificação de interfaces fere um dos princípios básicos da orientação a objetos, que diz que interfaces são imutáveis.

A interface *IAppServer* possui 7 métodos, e apenas 7, que fazem basicamente tudo o que você precisa em um servidor de aplicação, como solicitar dados, efetuar atualizações etc. A **Listagem 1** apresenta a referida interface.

Note que cada interface tem um GUID (*Global Unique Identifier - http://en.wikipedia.org/wiki/Globally_Unique_Identifier*), um identificador único de 128 bits que identifica a interface globalmente. Aqui o termo global significa mundo, é praticamente impossível existirem dois GUIDs iguais no globo.

Note também alguns importantes métodos na interface, como o *AS_GetProviderNames*. É esse método que é chamado, quando você baixa a lista *dropdown* da propriedade *ProviderName* de um *ClientDataSet* para ver os *DataSetProviders* disponíveis no servidor de aplicação ou locais. O *AS_GetParams* é acionado, quando você dá um clique de direita no *ClientDataSet* e chama o *FetchParams*, para solicitar que parâmetros de instruções SQL sejam passados na aplicação cliente (ou é acionado quando o método *FetchParams* do *ClientDataSet* é chamado em *runtime*).

*AS_Execute* é utilizado quando a propriedade *CommandText* do *ClientDataSet* é utilizada, ideal para passar instruções SQL diretamente da aplicação cliente. *AS_ApplyUpdates* é auto-explicativo, chamado quando atualizações são feitas no *ClientDataSet* e a seguir o método *ApplyUpdates* do mesmo componente é chamado. *AS_GetRecords* é invocado quando você chama por exemplo o método *Open* do *ClientDataSet*, para obter registros de uma tabela via servidor de aplicação ou localmente.

*DataRequest* é um método opcional para obter dados de um servidor de aplicação sem a necessidade da utilização de um *DataSetProvider* ligado a um *DataSet*, ideal no COM+ para não ser necessária a criação de métodos de negócio adicionais.

Note ainda que a interface *IAppServer* herda de *IDispatch*, interface que por sua vez herda de *IInterface* (a *IUnknown* renomeada). *IDispatch* é responsável por despachar chamadas remotas ao servidor de aplicação. Outro ponto notável é que todos os métodos estão com a convenção de chamada *safecall*, o que faz com que qualquer exceção levantada no servidor de aplicação seja empacotada e devolvida para o cliente. Caso contrário, se uma mensagem de erro fosse levantada no servidor e não no cliente, faria como que a solução inteira travasse, porque obviamente não existe um usuário operando o servidor de aplicação para pressionar o botão de Ok de uma exceção.

Como você pode ver, uma interface é apenas um mecanismo de definição, uma espécie de contrato que ambas as partes (cliente e servidor) devem conhecer e jogar suas regras. Ela não tem implementação (por isso mesmo é uma interface).

Quando formos criar nossos objetos COM+, uma nova interface será criada, descendente de *IAppServer*, onde poderemos incluir nossos métodos personalizados além dos 7 já pré-definidos. Quem implementa a interface é nosso objeto COM+, também conhecido como *CoClass*.

## Vantagens do desenvolvimento multicamadas

Agora que conhecemos um pouco mais da arquitetura do *DataSnap* e *IAppServer*, vamos enumerar as principais vantagens do desenvolvimento multicamadas:

*Centralização* – todas as regras de negócio e acesso a banco de dados ficam



Clube101.indb 28 10.11.08 15:53:02

centralizadas no servidor de aplicação. Isso quer dizer que uma mudança em um requisito, que exija a mudança em algum código, não fará com que todos os clientes precisem ser recompilados e redistribuídos através da rede;

*Modularização* – justamente por sua natureza multicamadas, aplicações desse tipo são mais fáceis de manter e atualizar, pois funcionam como módulos;

*Independência de BD* – o cliente não enxerga o BD, aliás, nem sabe que tipo de BD está sendo utilizado em dado momento. Isso significa que você pode ter uma mesma solução adaptável a diferentes servidores de banco de dados, como *Firebird, SQL Server, MySQL, DB2, Oracle* etc. O BD pode inclusive ser um mero repositório de tabelas e relacionamentos, visto que a maioria do código para tratamento de dados pode ser escrito no servidor de aplicação, garantindo uma maior independência;

*Performance* – aplicações multicamadas foram feitas para serem rápidas, principalmente ao se utilizar o protocolo COM+;

*Escalabilidade* – aplicações multicamadas são mais escaláveis, ou seja, não perdem performance quando um número maior de usuários conecta a aplicação, muito disso tendo em vista a natureza *state-less* do COM+;

*Economia de licenças* – alguns bancos de dados exigem o pagamento de licenças extras para instalação de bibliotecas de conexão cliente. No *DataSnap* - que possui *royalts free* para distribuição de clientes desde o Delphi 7 -, as bibliotecas clientes ficam somente no servidor de aplicação, o que pode economizar licenças. Podemos ter 20 usuários simultâneos, por exemplo, usufruindo do mesmo cliente de banco (no caso do *Firebird* a biblioteca *fbclient.dll*). Soma-se a isso a facilidade de instalação do cliente, pois alguns bancos possuem clientes com centenas de MB, imagine instalar isso em cada estação cliente que acessará o servidor. No *DataSnap*, o cliente é só um EXE e pronto.

## Vantagens do desenvolvimento COM+

Vamos agora enumerar algumas das vantagens ao se utilizar o COM+ como protocolo. As principais características do COM+ são gerenciamento, transações, escalabilidade, *pooling* e independência de cliente:

*Gerenciamento* – O COM+ oferece um console para gerenciamento dos componentes, onde se pode gerenciar pacotes (coleções de componentes), instalar e registrar novos componentes, visualizar interfaces, métodos e instâncias ativas, e configurar modelos de transação. Desde o Delphi 6, para facilitar o gerenciamento de componentes COM+, há um componente chamado *COMAdminCatalog*, da paleta COM+. Com ele, sua aplicação pode se tornar um cliente para os componentes COM+ do sistema. Você pode, por exemplo, escrever um gerenciador próprio para o COM+ dentro de sua aplicação, ou instalar componentes automaticamente, sem precisar do console do COM+;

*Transações* – Com o COM+, você pode realizar transações em nível de objetos, em vez de usar apenas transações de banco de dados. Isso permite que métodos chamados em vários objetos diferentes sejam colocados no contexto de uma única transação, mesmo que os objetos não façam acesso direto a bancos de dados. Há suporte também a transações distribuídas, envolvendo vários bancos de dados. O mecanismo responsável por este gerenciamento de transações chama-se *MSDTC* (*Microsoft Distributed Transaction Coordinator*).

*Escalabilidade* – O COM+ permite que se aumente drasticamente o volume de aplicações concorrentes sem haver perda de desempenho na camada de negócio, tudo graças ao mecanismo de *pooling* de objetos e ao modelo *state-less*;

*Pooling* – O COM+ pode manter 5 instâncias de um objeto atendendo a mil clientes, por exemplo. É isto que o torna tão escalável. Isso é feito através do mecanismo *state-less* e do *pooling* de objetos. Quando um cliente faz uso de um objeto, é o COM+ que gerencia como sua instância se comportará na memória. Com isso, ele pode liberar um objeto sem o conhecimento do cliente. Mas, e quando o cliente chamar um método de um objeto e ele não existir? O COM+ instancia-o novamente ou usa um objeto que já esteja no *pool*. Com o mecanismo *state-less*, objetos não mantêm o estado: eles não "se lembram" das últimas operações realizadas pelo cliente. Isso torna possível otimizar a instanciação dos objetos e o acesso a eles;

*Independência de cliente* – Outro aspecto a se observar é o fato de você poder utilizar um objeto COM+ feito em Delphi a partir de outras linguagens, como VB, ASP, C++, .NET etc. Com isso, você pode construir um poderoso *middleware* em *Delphi* para distribuir objetos a serem chamados em código ASP, por exemplo. Aliás, é muito comum desenvolvedores *Delphi* construírem componentes COM usando recursos da VCL, para serem utilizados em páginas ASP ou ASP.NET.

| Componente | Name | Propriedades |
|---|---|---|
| SQLConnection | SQLConnection | LoginPrompt=False |
| SQLQuery | qryDepartment | SQLConnection=SQLConnection<br>SQL= select * from DEPARTMENT<br>Active=True |
| DataSource | dsDepartment | DataSet=qryDeparment |
| DataSetProvider | dspDepartment | DataSet=qryDepartment |
| ClientDataSet | cdsDepartment | ProviderName=dspDepartment |
| SQLQuery | qryEmployee | SQLConnection=SQLConnection<br>SQL=select * from EMPLOYEE where DEPT_NO=:DEPT_NO<br>DataSource=dsDepartment |

**Tabela 1**. Componentes de acesso a dados do DataModule

## Uma aplicação Client/Server

Agora que conhecemos um pouco mais sobre o *DataSnap* e COM+, vamos construir um exemplo prático. Aqui utilizarei o *Delphi 7*, mas fique à vontade para utilizar outra versão, como o *Delphi 2007/2009*. Inicie uma nova aplicação *VCL Win32* e salve todos os arquivos em uma pasta chamada *Client*. Através do menu *File>New* crie um novo *DataModu-*



Clube101.indb 29 10.11.08 15:53:05

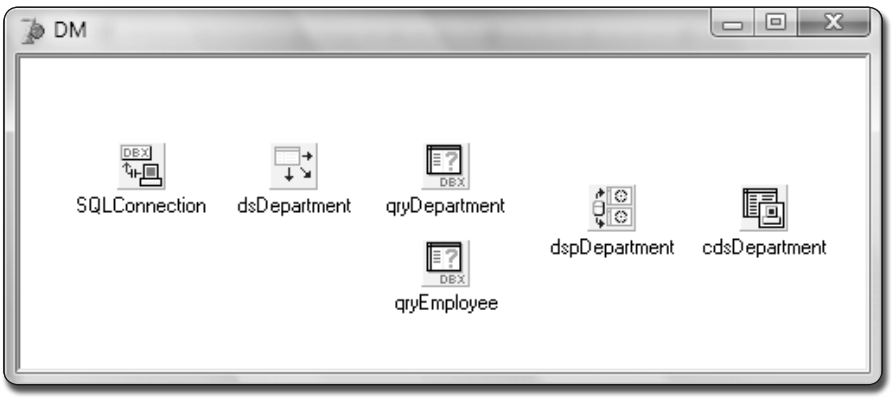



**Figura 2**. DataModule da aplicação

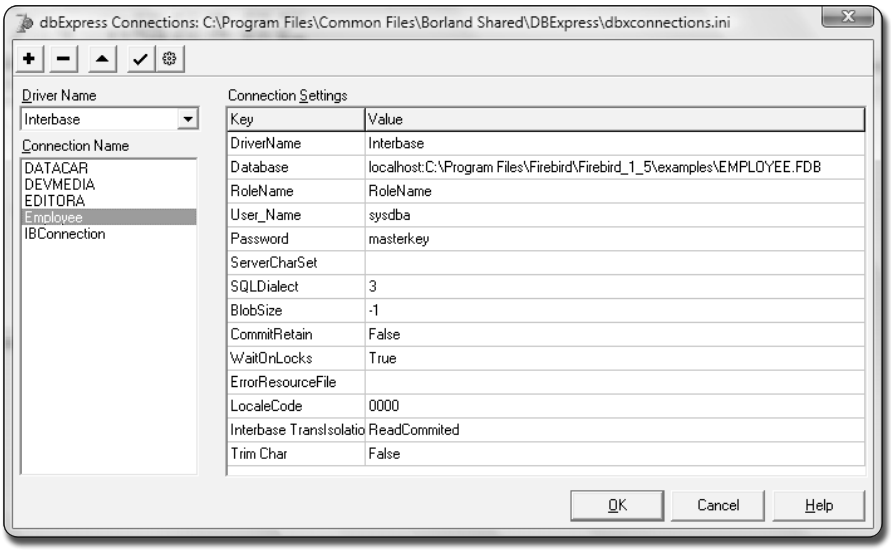



**Figura 3**. Configurando a conexão

*le*, dando a ele o nome de *DM*. Usando as paletas *dbExpress* e *Data Access* coloque e configure os componentes descritos na **Tabela 1.** Ao final seu *DM* deve estar semelhante ao da **Figura 2**.

Dê um duplo clique no *SQLConnection*, no editor clique no botão +, selecione o driver *Interbase* e em *Name* digite *Employee*. Configure os parâmetros de conexão como mostrado na **Figura 3.** Note a propriedade *Database* que aponta para o banco *Employee.fdb* que acompanha o *Firebird*. Clique em Ok. Na propriedade *VendorLib* do *SQLConnection* digite *fbClient.dll*, que é a biblioteca cliente nativa do *Firebird*. Certifique-se que essa DLL esteja no *path* do *Windows*, ela pode ser encontrada no diretório de instalação do *Firebird*.

Vá na propriedade Params do qryEmployee e configure o parâmetro DEPT_NO como sendo ParamType=ptInput e DataType=ftInteger.

Vamos entender o que foi feito até aqui. Os componentes colocados configuram uma relação *master/detail* entre as tabelas *Department* (departamentos) e *Employee* (empregados) do banco. Ou seja, um departamento possui vários empregados. O interessante dessa relação é que usamos um *DataSource* apontando para a *query* de departamentos, e a propriedade *DataSource* (que pode parecer estranha estar presente em um *DataSet*) aponta para o *dsDepartment*. Com isso, o *dbExpress* entende que há uma relação mestre/detalhe entre as duas tabelas, e o *DataSetProvider* pode empacotar o conteúdo de ambas as tabelas em um único *DataPacket*.

Dê um duplo clique em *qryDepartment* e no editor aperte *Ctrl+A* e confirme em

## Nota do DevMan

*DataPackets* são pacotes de dados que trafegam através das chamadas de procedimento remoto do *DataSnap*. Por exemplo, os dados de uma tabela, no servidor de aplicação, são empacotados em um formato específico (binário ou XML) para serem enviados pela rede ao cliente. Esse pacote é também conhecido como *Data*. No *Delphi*, quem cuida de criar o *DataPacket* é a biblioteca MIDAS.DLL. O segredo do empacotamento, por ser criar pacotes pequenos e otimizados, é um segredo que a equipe do *Delphi* guarda até hoje. Os pacotes também podem estar em formato XML, no caso de aplicações SOAP. Para trabalhar com intercâmbio de dados XML do *DataSnap* com plataformas diferentes, usamos o utilitário *XML Mapper* (veja edições 36 e 100). Quando alterações são feitas no *ClientDataSet* e repassadas de volta ao servidor de aplicação, elas são novamente empacotadas, num *DataPacket* conhecido como *Delta*.

## Nota do DevMan

*TFields* são objetos que representam campos de um *DataSet*. É possível utilizá-los de duas formas, ou acessando o método *FieldByName* ou a collection *Fields* do *DataSet*, para obter um objeto *TField* (classe base de todos os *TFields* do Delphi) ou ainda adicionando os campos fortemente tipados (*TIntegerField*, *TStringField*) no próprio *DataSet*, dando um duplo clique sobre ele.

## Nota do DevMan

O que significa o parâmetro "0" (zero) passado no *ApplyUpdates*? Quando fazemos atualizações no *ClientDataSet*, nada é aplicado no servidor até você chamar *ApplyUpdates*. Tudo fica em uma cache local. Normalmente um registro é atualizado no BD logo após o POST, mas isso não é obrigatório. O zero do *Apply* significa que a operação deve falhar caso algum erro seja percebido na atualização de pelo menos um registro da *cache delta*. Para aplicar a maior quantidade de registros possíveis, passamos -1. Os erros originados desse *Apply* devem ser tratados no evento *OnReconcileError* do *ClientDataSet*.



Clube101.indb 30 10.11.08 15:53:12

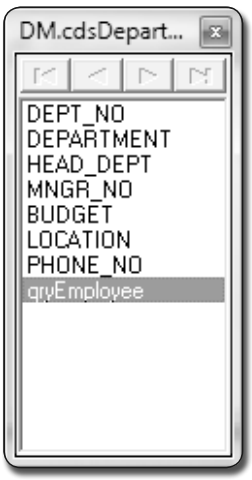



**Figura 4**. DataSetField

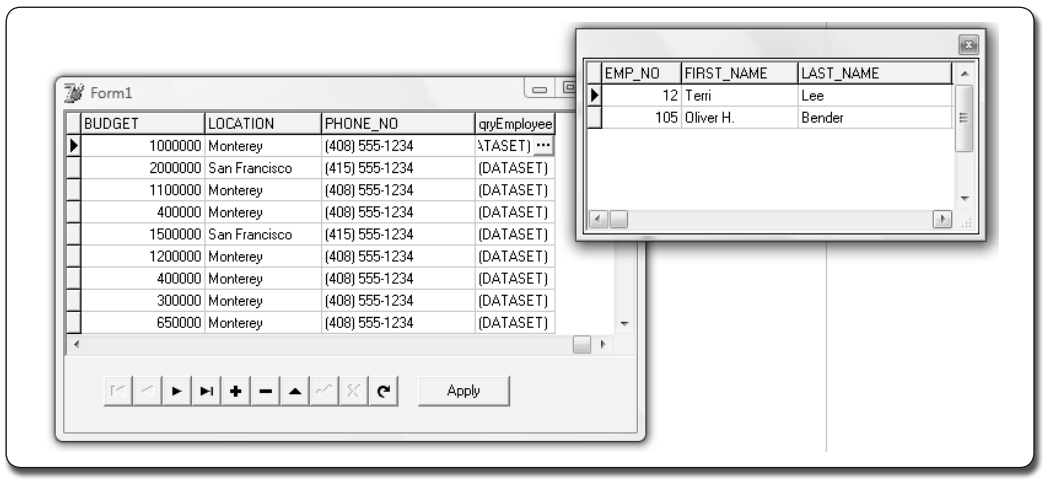



**Figura 5**. Aplicação em funcionamento

Ok para adicionar os campos *TFields* ao *DataSet*. Repita o mesmo procedimento para o *qryEmployee* e *cdsDepartment*.

Ao adicionar os *TFields* no *ClientDataSet*, você deve ter notado que além de todos os campos da tabela *Department* (**Figura 4**), há também um campo com o nome da *query detail* da relação. Esse é um campo que representa nada mais nada menos que a tabela inteira de empregados. Esse recurso é conhecido no Delphi como *DataSetFields*. Além disso, o *DataSnap* é capaz de empacotar cada departamento juntamente com seus respectivos empregados (daí a necessidade de um único *DataSetProvider* para ambas as tabelas).

## Criando a interface

No formulário principal da aplicação coloque um *DataSource* (*Data Access)* e um *DBGrid (Data Controls)*. Aponte o *DBGrid* para o *DataSource* através da propriedade de mesmo nome. Clique em *File>Use Unit* e selecione a unit do *DM* (que deve ser a *unit2*). Aponte o *DataSource* do *form* para o *cdsDepartment*. Coloque também um *DBNavigator (DataControls)* e aponte-o para o *DataSource* do *form*. Coloque um *Button* e no seu evento *OnClick* digite:

```
procedure TForm1.ApplyClick(Sender: TObject);
begin
  DM.cdsDepartment.ApplyUpdates(0);
end;
```

Dê um duplo clique no *form* e no seu evento *OnCreate* digite:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DM.cdsDepartment.Open();
end;
```

Antes de executar a aplicação, clique em *Project>Options* e certifique-se que o *DM* seja criado antes do *form*. Pressione *F9* para executar a aplicação (**Figura 5**).

Se quiser ocultar a palavra *(DATASET)* que aparece como conteúdo do último campo, basta selecionar o campo *DataSetField* no editor de campos do *ClientDataSet* e no seu evento *OnGetText* digitar:

```
procedure TDM.cdsDepartmentqryEmployeeGetTex
t(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
   if Sender = cdsDepartmentqryEmployee then
     Text := '';
end;
```

## Como funciona o dbExpress e DataSnap

Vejamos agora como funciona do *DataSnap* / *dbExpress*. Mesmo que você esteja utilizando uma aplicação *client/server* normal, o funcionamento a seguir explicado ainda vale. Isso porque o *dbExpress* local funciona de forma muito semelhante ao *DataSnap* multicamadas, por fazer uso dos componentes *ClientDataSet* e *DataSetProvider*. A **Figura 6** demonstra esse funcionamento. Considerando uma operação simples de *select* e atualização, temos os seguintes passos executados:

1 – O método *Open* do *ClientDataSet* é chamado. Isso faz internamente uma chamada a *AS_GetRecords* ser invocada. Nessa chamada a *IAppServer*, o nome do *provider* indicado na propriedade *ProviderName* é passado ao servidor, que se encarrega de localizar o componente *DataSetProvider* associado;

2 – No servidor de aplicação, o *DataSetProvider* dispara importantes eventos, como *BeforeGetRecords*, caso você queira fazer alguma operação antes dos dados serem obtidos a partir do *engine* de conexão;

3 – O *DataSetProvider* se comunica com o *engine* de conexão, nesse caso o *dbExpress*, para obter os dados o *select*. Ele vai abrir um cursor unidirecional de dados indicado pela sua propriedade *DataSet*. Essa comunicação com o *DataSet* associado é feito também através de uma interface, chamada *IProviderSupport*. Como podemos ver, todos os *DataSets* do Delphi devem implementar essa interface para dar suporte ao empacotamento e reconciliação de dados. Comprovamos isso pela declaração da classe *TDataSet* na unit *DB.pas* da *VCL*:

```
TDataSet = class(TComponent, IProviderSupport)
```

A interface *IProviderSupport* é mostrada na **Listagem 2**. Nela podemos observar importantes métodos como *PSExecute* e *PSExecuteStatement*, encarregado de executar comandos SQL, *PSGetParams*, para recuperar uma lista de parâmetros para ser enviada do *DataSet* servidor ao *ClientDataSet* local, *PSUpdateRecord* para atualizar um *DataPacket* de dados etc. Esses métodos são todos declarados como virtuais em *TDataSet*, o que significa que o *DataSet* descendente, como o *TSQLDataSet*, deve sobrescrevê-los;

4 – O *DataSetProvider*, via interface *IProviderSupport*, chama o método *Open* do *DataSet* que pode estar fechado, que por sua vez abre a conexão com o BD



Clube101.indb 31 10.11.08 15:53:16

**Listagem 2**. Interface IProviderSupport

```
IProviderSupport = interface
['{7AFt8F684-0660-47B5-A1B3-E168D2ACB908}']
  procedure PSEndTransaction(Commit: Boolean);
  procedure PSExecute;
  function PSExecuteStatement(const ASQL: string; AParams: TParams;
    ResultSet: Pointer = nil): Integer;
  procedure PSGetAttributes(List: TList);
  function PSGetDefaultOrder: TIndexDef;
  function PSGetKeyFields: string;
  function PSGetParams: TParams;
  function PSGetQuoteChar: string;
  function PSGetTableName: string;
  function PSGetIndexDefs(IndexTypes: TIndexOptions = [ixPrimary..ixNonMaintained]): TIndexDefs;
  function PSGetUpdateException(E: Exception; Prev: EUpdateError): EUpdateError;
  function PSInTransaction: Boolean;
  function PSIsSQLBased: Boolean;
  function PSIsSQLSupported: Boolean;
  procedure PSReset;
  procedure PSSetParams(AParams: TParams);
  procedure PSSetCommandText(const CommandText: string);
  procedure PSStartTransaction;
  function PSUpdateRecord(UpdateKind: TUpdateKind; Delta: TDataSet): Boolean;
```

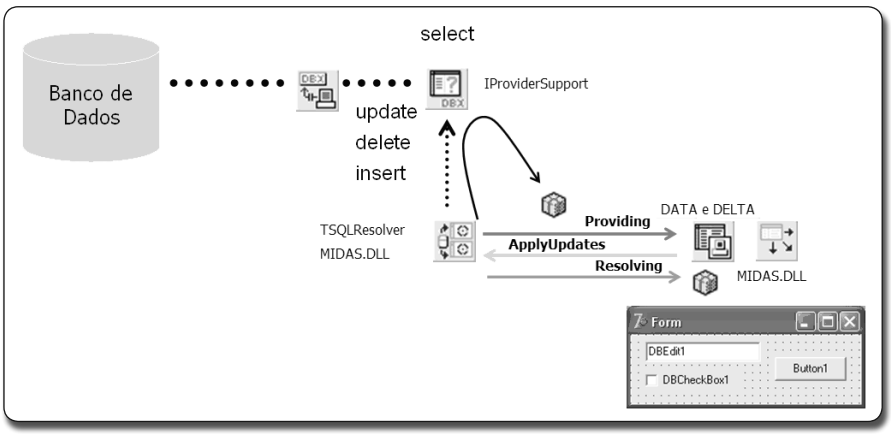



**Figura 6**. Funcionamento do dbExpress e DataSnap

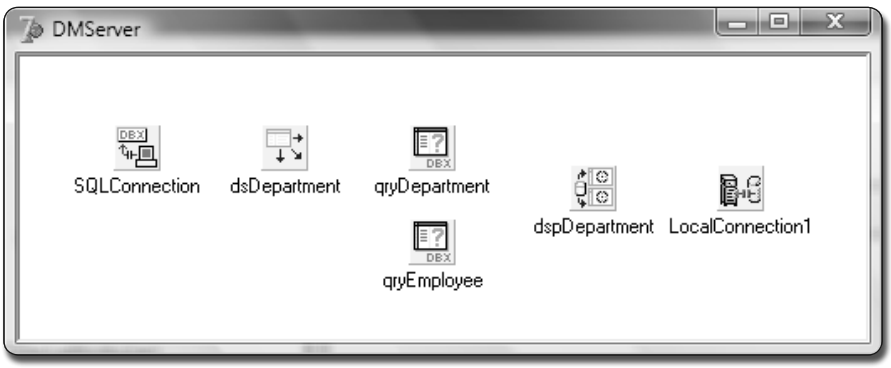
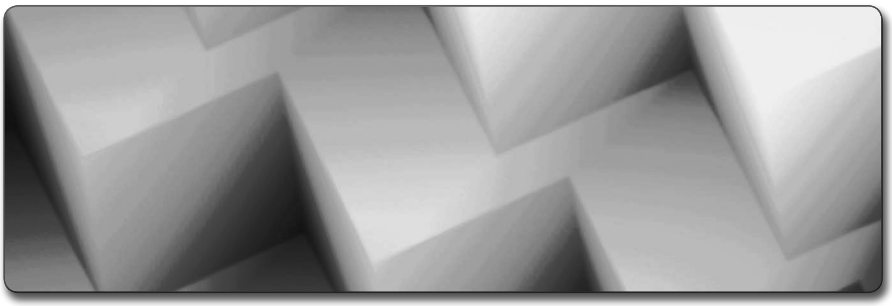



**Figura 7**. DMServer simula um AppServer local



via *SqlConnection*. Um cursor de dados é aberto no servidor, e o *DataSetProvider* inicia a varredura desse cursor (na verdade é um *while not eof* no *DataSet*). Ele vai então criando um pacote de dados (*DataPacket*) chamado *Data*. Internamente, quem cria esse pacote *Data* é a classe *TCustomPacketWriter* da unit *Provider.pas*, auxiliada pela biblioteca MIDAS.DLL. A comunicação com a MIDAS.DLL também é feita através de interfaces, como de costume, como pode ser visto na unit *IDSWriter.pas* da *VCL*;

5 – Criado o *DataPacket*, o *Data* é enviado através do *DataSnap* para o respectivo *ClientDataSet*, que se encarrega de guardar em sua propriedade *Data* os dados oriundos do *AppServer*. Os dados são exibidos ao usuário;

> **Nota**: O processo de comunicação, troca de dados e serialização de dados entre diferentes processos é conhecido como *marshalling*.

6 – O usuário localmente faz várias atualizações em vários registros do *DataPacket*, que vão ficando guardados em memória, em um outro pacote chamado *Delta*. Quando o método *ApplyUpdates* é chamado, há uma comunicação com *IAppServer* através do método *AS_ApplyUpdates*. O *Delta* é passado então ao

## Nota do DevMan

Como o *TSQLResolver* sabe montar as instruções SQL de atualização? Para isso o *DataSetProvider*, através da interface *IProviderSupport*, consulta a propriedade *ProviderFlags* de cada campo *TField* do *DataSet* indicado na propriedade de mesmo nome, normalmente um *SQLDataSet* ou *SQLQuery*. Nesse *TField*, o campo indica nas opções *pfInWhere, pfInKey e pfInUpdate* se deve ser colocado nas cláusulas *update/insert/delete e where* de cada comando de atualização gerado. Normalmente, o campo chave tem *pfInKey* e *pfInWhere* igual a *True*. Os demais campos têm o *pfInWhere* como *False*. Campos oriundos de *joins*, como descrições, normalmente não são atualizados (somente a *Foreign-key*), então têm o *pfInUpdate* como *False*.



Clube101.indb 32 10.11.08 15:53:58

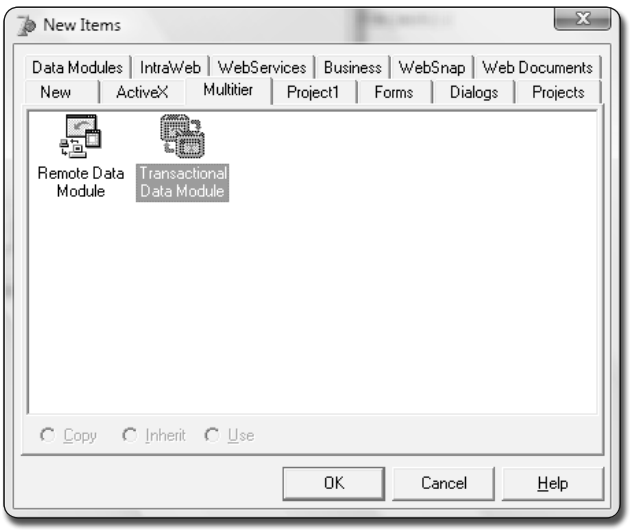



**Figura 8**. Criando o servidor de aplicação

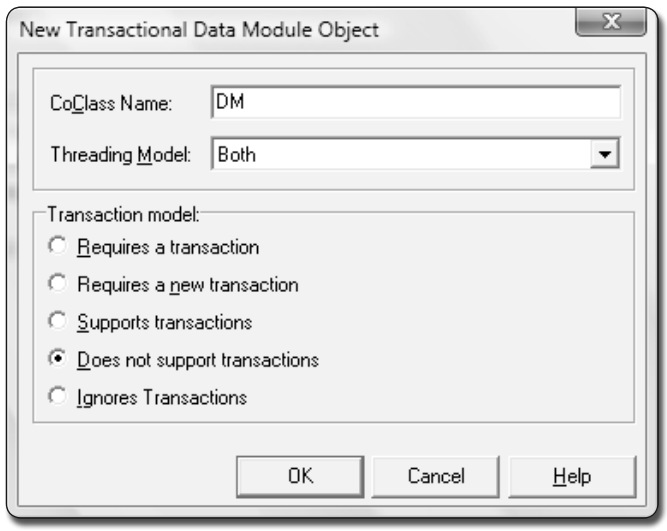



**Figura 9**. Parâmetros do servidor

servidor de aplicação e recebido pelo *DataSetProvider*;

7 – O *DataSetProvider* inicia então um processo chamado *Resolving*, que consiste em desmembrar os registros do *Delta* em instruções SQL de *insert*, *update* e *delete*. Antes de cada operação, ele chama o importante evento *BeforeUpdateRecord*, indicando no parâmetro *UpdateKind* o tipo de atualização que está sendo feita. É nesse ponto que podemos efetuar uma validação por exemplo, ou resolver uma atualização com *join* onde mais de uma tabela sofre um *update/insert/delete*. Novamente, essa comunicação é feita com o *DataSet* do *dbExpress* através da interface *IProviderSupport*. A geração dos comandos SQL de atualização é feita automaticamente. Salvo várias exceções, no *dbExpress* você raramente escreve *updates*, *inserts* e *updates*, apenas o *Select*. Isso se deve graças a classe *TSQLResolver* da unit *provider.pas*, que monta as instruções SQL no momento do *Resolving*. Outros importantes eventos, como *BeforeApplyUpdates*, são chamados no *DataSetProvider*;

8 – O *SqlDataSet* tenta então executar as instruções SQL passadas via *IProviderSupport*. Se houver algum erro gerado no SGBDR, como a violação de uma *ForeignKey*, um novo processo é desencadeado, chamado *Resolving*. Os registros problemáticos são devolvidos ao *ClientDataSet* para serem corrigidos, procedimento feito no seu evento *OnReconcileError*.

## DataSnap Local

Podemos começar nossa migração de *client/server* para *DataSnap* usando um *AppServer* local. Crie um novo *DataModule* chamado *DMServer*. Agora dê um *Ctrl+X* nos componentes *dbExpress* que estão no DM (*SQLQuerys* e *SQLConnection*) e também *DataSource* e *DataSetProvider*, colando-os no *DMServer*. Da paleta *DataSnap* coloque um componente *LocalConnection*. Seu *DMServer* deve estar semelhante ao da **Figura 7**.

### Nota do DevMan

Por que a propriedade *ProviderName* do *ClientDataSet* é uma string e não uma referência para um objeto, como a propriedade *DataSet* de um *DataSource*? Uma referência a objeto não faz sentido em uma aplicação multicamadas, pois um objeto estará tentando referenciar um endereço de memória inválido rodando em outro processo em outra máquina. Por esse motivo *ProviderName* é uma string, que é passada pelo método *AS_GetRecords* de *IAppServer* quando o método *Open* do *ClientDataSet* é chamado. Por esse motivo, se você alterar o nome de um *DataSetProvider* depois de ter configurado o *ProviderName*, deverá manualmente refazer a referência, pois o IDE não toma conhecimento da mudança como o que acontece com ponteiros diretos para objetos.

Volte ao *DM* e clique em *File>Use Unit*, escolhendo a *unit* do *DMServer* (que deve ser a *unit3*). Agora selecione o *ClientDataSet* e em *RemoteServer* escolha *DMServer.LocalConnection1*e mantenha o valor de *ProviderName*. Clique em *Project>Options* e certifique-se que a ordem de criação dos itens seja *DMServer, DM* e *Form1*. Execute e teste a aplicação.

**Nota**: Uma opção ao uso do *LocalConnection* é chamar o método *SetProvider* do *ClientDataSet* e passar o *DataSetProvider* como parâmetro, assim:

*cdsDepartment.SetProvider(DMServer.dspDepartment)*

Configurar a propriedade *ProviderName* manualmente não funciona.

## DataSnap em ação: criando o servidor

Vamos iniciar o desenvolvimento multicamadas propriamente dito. Clique em

### Nota do DevMan

O que é uma CoClass? CoClass é uma classe COM+ que implementa uma interface COM. Por exemplo, um TransactionalDataModule ou um RemoteDataModule são ambos CoClasses.



Clube101.indb 33 10.11.08 15:54:01

*File>New>Other* e na aba *Multitier* escolha *Transactional Data Module* (**Figura 8**). Na mensagem que aparece, informando que será criado um projeto *ActiveX*, clique em Ok. Na próxima tela (**Figura 9**), dê o nome de *DM* para a *CoClass* e escolha o tipo *Both* para *Threading Model*. Isso é necessário para que possamos usar o recurso de *Object Pooling* visto a seguir. Clique em Ok e salve todos os arquivos em uma pasta chamada *Server*. Veja que foi criado um *DataModule*.

Agora abra apenas o *DMServer* da pasta *Client*, dê um *Ctrl+X* nos componentes que cole no *DataModule* da aplicação *DataSnap*. Retire o *LocalConnection* porque ele não será mais necessário aqui.

Uma boa dica ao se trabalhar com aplicações *DataSnap* é criar um grupo de projetos (*Project Group*). Para isso, vá até a opção *View>Project Manager*, dê um clique de direita sobre o grupo de projetos e escolha *Add Existing Project*, escolhendo o projeto cliente. Salve o grupo com a extensão BPG. Agora ambos os arquivos dos projetos po-

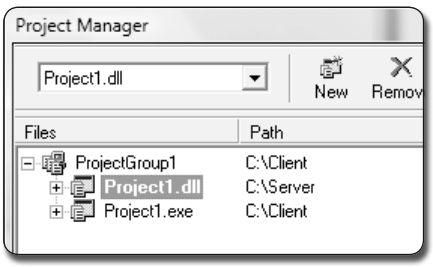



**Figura 10**. Grupo de projetos

dem ser facilmente acessados, (**Figura 10**), não é necessário fechar a aplicação servidora para trabalhar com um formulário da aplicação cliente, por exemplo. Com o servidor selecionado, clique em *Project>Build* para compilar a DLL do servidor COM+.

## Registrando o servidor

Agora precisamos registrar o servidor de aplicação para que possa ser acessado por aplicações clientes. No *Painel de Controle>Ferramentas Administrativas*, acesse os *Serviços de Componentes*. Usuários do Windows Vista devem digitar no *Executar*:

```
C:\windows\system32\comexp.msc
```

O que você está vendo é o gerenciador do catálogo do COM+. Abra a árvore até encontrar o item referente ao seu computador, clique de direita e *COM+ Applications* e escolha *New>Application*. No assistente clique em *Next*, depois clique em *Create an empty application* (**Figura 11**), dê um nome para a aplicação (*ClubeDelphi*) e finalize o assistente.

De volta ao gerenciador, vá até a aplicação criada e clique de direita em *Components*, escolhendo a seguir *New>Component*. No gerenciador que abre clique em *Next* e depois *Install new component(s)* (**Figura 12**). Escolha então a TLB (*Type Library*, discutida a seguir) do projeto servidor e finalize o assistente. Seu componente deve estar registrado conforme a **Figura 13**.

> **Dica**: você pode usar esse gerenciador para administrar objetos de outras máquinas, bastando para isso registrar o computador remoto.

## Testando o cliente

Agora na aplicação cliente remova o *DMServer*, pois não usaremos mais um *DataModule* local, mas sim o *Transactional Data Module* da aplicação servidora. Para isso acesse o *Project Manager*, clique de direita sobre a *unit* desse *DataModule* (que deve ser a *unit3*) e escolha *Remove from Project*. Agora, do *DM* remova o *uses unit3* da cláusula *uses* (sugiro também apagar o *DMServer* do disco para não haver dúvidas que vamos usar o *DataModule* remoto e não o local).

Agora, no *DataModule* da aplicação cliente, onde deve ter sobrado apenas um *ClientDataSet*, coloque um *DCOMConnection* da guia *DataSnap*. Em *ServerName* escolha o nosso *Transactional DataModule*, que deve ser algo como *Project1.DM*. Note que a propriedade *ServerGuid* já é configurada automaticamente. Aponte o *RemoteServer* do *ClientDataSet* para o *DCOMConnection* e execute a aplicação. Note que tudo funciona perfeitamente, só que agora usando 3 camadas.

Nesse momento o objeto será levantado no servidor. A nossa DLL do *Transactional DataModule* ficará hospedada em um processo servidor chamado *DLLHost.exe*. Você pode ver esse processo em

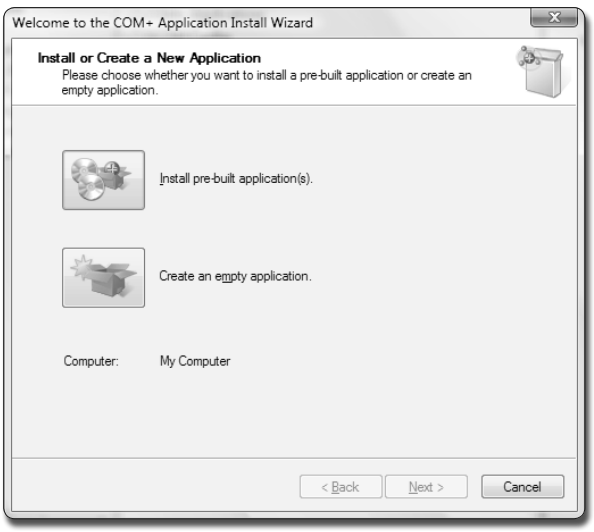



**Figura 11**. Criando uma aplicação COM+

**Figura 12**. Instalando componentes



Clube101.indb 34 10.11.08 15:54:03

execução pelo *Task Manager* do Windows (ou ainda pelo utilitário bem melhor chamado *Process Explorer*). Se você tentar recompilar a aplicação servidora, receberá um erro dizendo que o arquivo DLL não pode ser criado:

```
[Fatal Error]
Could not create output file 'Project1.dll'
```

Isso acontece porque a DLL está em execução, hospedada no *DLLHost* (**Figura 14**). Para tirá-la da memória, você precisa no catálogo do COM+ dar um clique de direita na aplicação (*ClubeDelphi*) e escolher *Shutdown* (*Desligar*). Note que o ícone que representa o objeto COM+ fica animado enquanto estiver em execução.

> **Nota**: Um bom atrevimento é finalizar o processo *DLLHost.exe* manualmente para recompilar o servidor.

## Object Pooling

Um recurso interessante do COM+ chama-se *Object Pooling*. Esse recurso permite que objetos sejam mantidos em memória no servidor após serem executados, e são reutilizados no momento de sua ativação. Como o COM+ é *state-less*, em algumas situações é vantajosos ativar o *Pooling* para que usuários, após terem utilizado objetos, deixem-os prontos e criados para serem utilizados por outros usuários.

Imagine uma conexão com banco de dados. Se para cada usuário que conectar no servidor de aplicação abríssemos uma conexão, isso poderia degradar a performance. A abertura da conexão com o BD é um dos recursos mais caros da programação. Então porque não guardar a conexão em *pool* (uma espécie de *cache*) e reutilizá-las para múltiplos usuários? Todo o tempo necessário para estabelecer uma conexão TCP/IP com o servidor de banco de dados e autenticar o usuário pode ser eliminado, visto que quando um usuário solicitar dados, já pegará uma conexão pronta do *pool*.

Vejamos como isso funciona na prática. No catálogo do COM+ dê um clique de direita no nome da aplicação e na aba *Pooling & Recycling* coloque o valor *10*

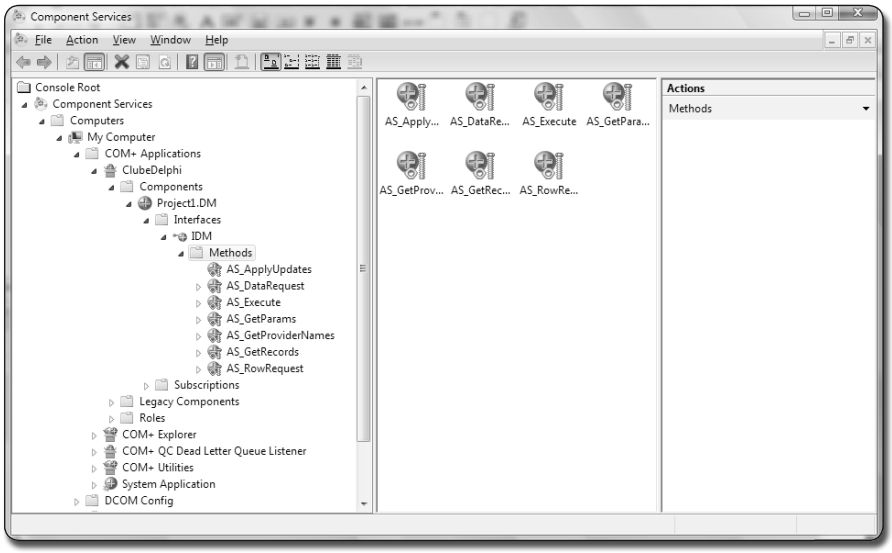



**Figura 13**. Componente instalado

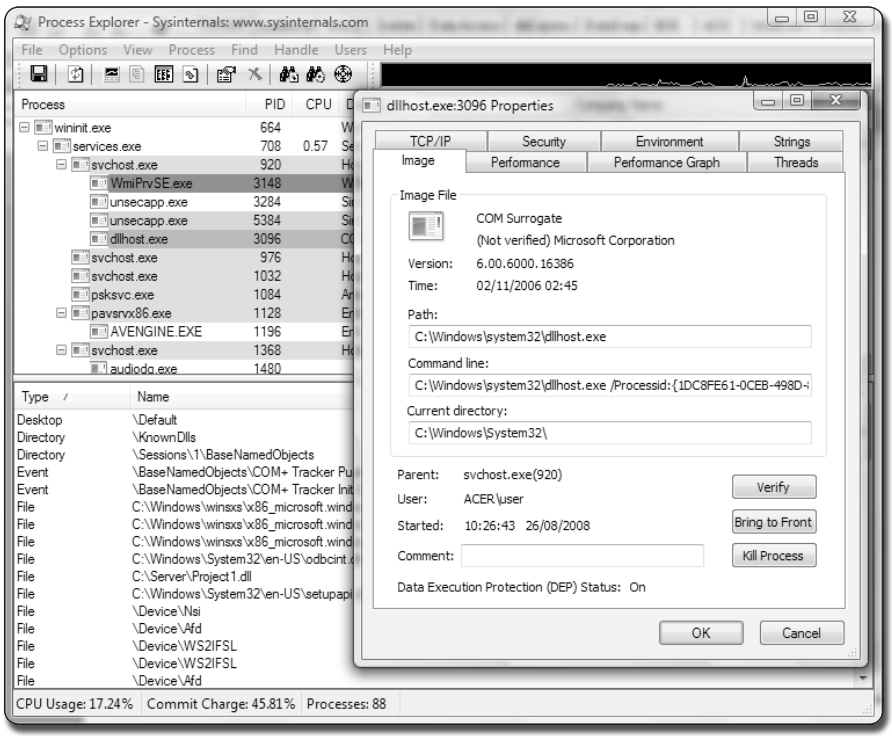
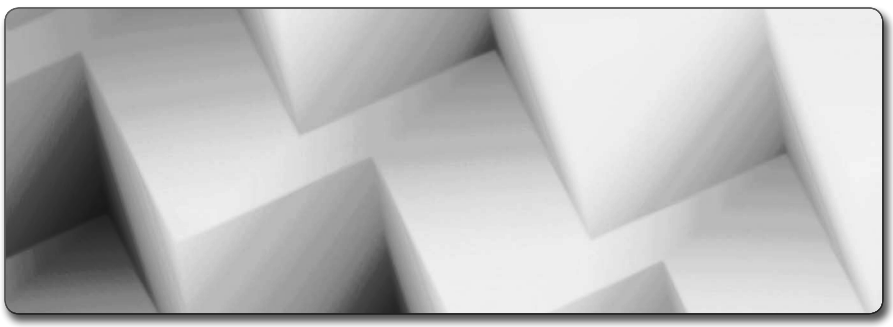



**Figura 14**. Processo DLLHost.exe hospeda nosso servidor COM+





Clube101.indb 35 10.11.08 15:54:46

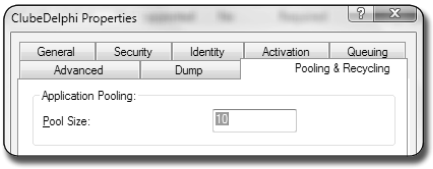



**Figura 15.** Ativando o Object Pooling na aplicação

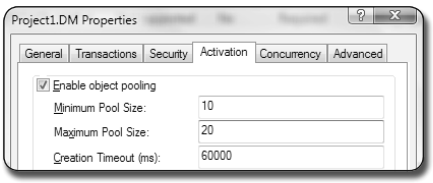



**Figura 16**. Ativando o Object Pooling no componente

(**Figura 15).** Esse passo é necessário para configurar o *Object Pooling* na aplicação.

O próximo passo é configurar o *Object Pooling* no componente. Para isso dê um clique de direita no componente (*Project1.DM*) e na aba *Activation* escolha *Minimum Pool Size=10* e *Max Pool Size=20* (**Figura 16**). Com isso estamos definindo que nosso *pool* terá no mínimo 10 componentes (leia-se *Transactional DataModule*) ativos e pode crescer até 20 conforme demanda. O *CreationTimeOut* indica em milésimos de segundo quanto tempo um objeto deve permanecer no *pool* se não for usado (*timeout*). Incremente esse valor de acordo com sua necessidade.

Agora algumas alterações são necessárias no *Transactional DataModule*. Defina a propriedade *KeepConnection* como *False* e *Connected* do *SQLConnection* como *True*. Isso fará com que a conexão fique aberta após ser usada, e quando o objeto for devolvido para o *pool*, não será necessário reabri-la quando um novo cliente solicitar dados do banco de dados. Isso acrescenta um ganho de performance imenso, visto que o tempo necessário para se estabelecer uma conexão TCP/IP, autenticar o usuário, conectar, não será necessário.

Para ver o *Object Pooling* em ação, no catálogo do COM+, selecione o item *Components* e a seguir clique no botão *Status*. Veja (**Figura 17**) que temos 10 objetos em *pool* (é necessário executar o cliente).

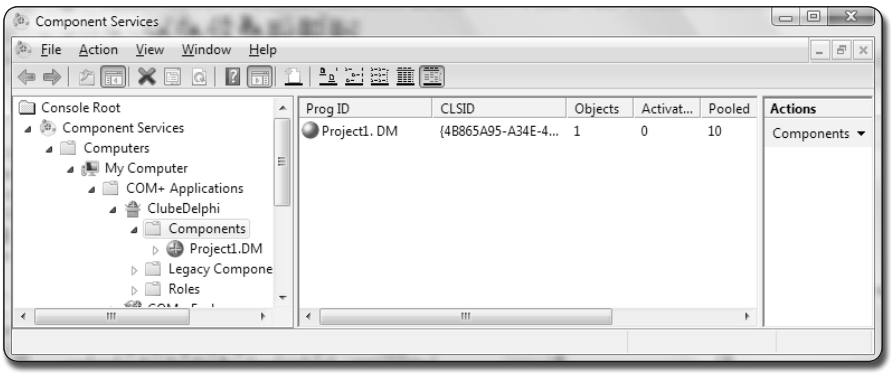



**Figura 17**. Object Pooling em ação

**Listagem 3.** Stored Procedure

```
create procedure AVG_SALARY (COD_DEPT integer)
returns (TOTAL numeric(15,2))
as
begin
   select
      avg (SALARY)
    from
       EMPLOYEE
    where
       DEPT_NO=:COD_DEPT
    into :TOTAL;
end
```

**Listagem 4.** Implementação do método remoto

```
function TDM.AVG_SALARY(DEPT_NO: Integer): Double;
begin
  SQLStoredProc1.ParamByName('COD_DEPT').AsInteger := DEPT_NO;
  SQLStoredProc1.ExecProc;
  result := SQLStoredProc1.ParamByName('TOTAL').AsFloat;
end;
```

**Listagem 5.** Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
 Label1.Caption := IntToStr(
   DM.DCOMConnection1.AppServer.AVG_SALARY
     (DM.cdsDepartmentDEPT_NO.AsInteger));
end;
```

## SimpleObjectBroker

Um recurso interessante do *DataSnap* é a realização de um *balanceamento de carga*. Digamos que sua aplicação multicamadas, por algum motivo, comece a perder escalabilidade. Você pode adicionar mais servidores replicados e assim ter múltiplos servidores de aplicação na mesma solução. No cliente, usamos um componente chamado *SimpleObjectBroker* para aleatoriamente selecionar um objeto COM+ a ser usado em uma operação.

> **Nota**: O catálogo do COM+ permite criar um pacote de instalação MSI para você replicar automaticamente um servidor para outro.

Abra o *DM* da aplicação cliente e nele coloque um *SimpleObjectBroker* da paleta *DataSnap*. Defina sua propriedade *LoadBalanced* como *True*, abra o editor da propriedade *Servers* e adicione o nome dos computadores servidores (**Figura 18**). Agora defina a propriedade *ObjectBroker* do *DCOMConnection* para *SimpleObjectBroker1*.

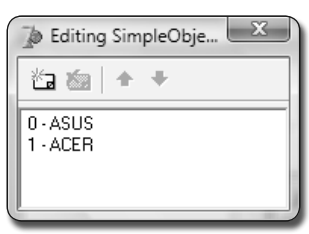



**Figura 18**. Balanceamento de carga no DataSnap



Clube101.indb 36 10.11.08 15:54:58

## Chamando métodos remotos

*IAppServer* já nos quebra um imenso galho ao permitir realizar todas as operações de banco de dados em apenas 7 métodos. Porém, normalmente, teremos outros métodos em nossa interface, como métodos de negócio. Com isso, estamos estendendo a interface *IAppServer*, adicionando métodos personalizados.

Imagine o caso de chamar uma *Stored Procedure*. O cliente não pode invocar uma SP diretamente do banco de dados porque isso fere os princípios do desenvolvimento em 3 camadas. A comunicação deve passar pelo servidor de aplicação. Nesse caso, nada melhor do que criar um método personalizado no servidor de aplicação e invocá-lo a partir do cliente. Esse método no servidor é que fará a comunicação com o *Firebird*.

Consideremos o seguinte cenário. Uma SP que precisa retornar a média salarial de todos os funcionários de um determinado departamento do banco *Employee.fdb*. Usando o *IBExpert* ou a sua ferramenta preferida para administração de banco de dados *Firebird*, crie a *Stored Procedure* da **Listagem 3**, que retorna a média salarial de todos os funcionários de um dado departamento.

Abra o *Transactional DataModule* na aplicação servidora e coloque um *SQLStoredProc*. Aponte para o *SQLConnection* através da propriedade de mesmo nome. Em *StoredProcName* aponte para procedure *AVG_SALARY* criada anteriormente.

No IDE do *Delphi*, clique em *Tools>Environment Options*. Em *Type Library>Language* escolha *PASCAL*. Isso afetará como os tipos de dados serão exibidos no editor da *Type Library*.

Clique em *View>Type Library*, dê um clique de direita em *IDM* e escolha *New>Method*. Dê o nome de *AVG_SALARY* para ele. Na guia *Parameters* coloque *Double* em *Return Type* e adicione um parâmetro chamado *DEPT_NO (integer)*, como mostra a **Figura 19.**

Se você abrir o arquivo da *Type Library* que está no projeto, verá que nossa interface *IDM* define agora o *AVG_SALARY*:

```
IDM = interface(IAppServer)
  ['{59D9816F-5FB2-45C8-9593-33A37E4FD171}']
  function AVG_SALARY(DEPT_NO: Integer):
    Double; safecall;
end;
```

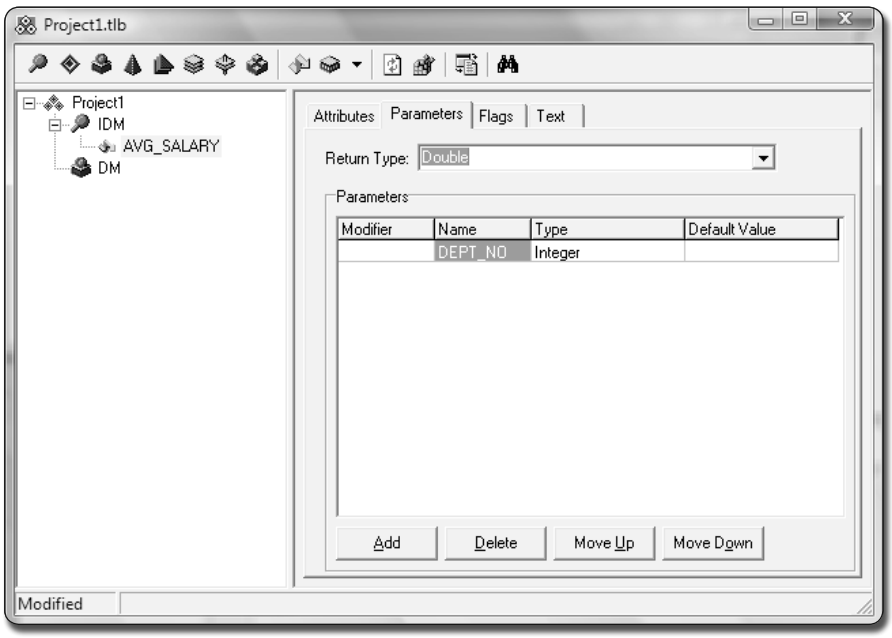



**Figura 19**. Método adicional na Type Library

Na *unit* do *Transactional DataModule* observe que o *Delphi* já declarou o método na *CoClass* e criou o cabeçalho da implementação. Resta-nos implementar então o método que chama a *Stored Procedure* no banco de dados e retornar o resultado para a aplicação cliente (**Listagem 4**).

Dê um *Build* no servidor. Na aplicação cliente coloque um botão em um *Label*. No evento *OnClick* do botão digite o código da **Listagem 5.** Execute a aplicação, selecione um departamento no *grid* e clique no botão para ver a média dos salários desse departamento (**Figura 20**).

Veja que usamos a propriedade *AppServer* do objeto de conexão *DataSnap* para invocar o método. *AppServer* é um tipo *variant*, o que significa que o compilador só vai reconhecer o verdadeiro significado das chamadas de métodos em tempo de execução, processo conhecido como *late-binding*. Veja por exemplo que podemos escrever qualquer coisa no lugar da chamada do método remoto, como mostra a **Listagem 6**, sendo que obteremos o erro em runtime da **Figura 21**. O efeito disso

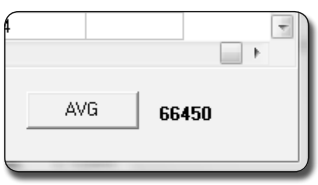



**Figura 20**. Chamando métodos remotos

### Nota do DevMan

*Type Library* é um nome auto-explicativo. Ela é uma biblioteca (em duplo formato - .PAS e .TLB) que define os tipos de um objeto servidor, além de seus métodos. A *Type Library* funciona como o "controle remoto" em uma comunicação multicamadas, um protocolo. Ambos, cliente e servidor, devem conhecer e concordar com os teremos definidos na *Type Library*.

### Nota do DevMan

*Variants* surgiram no Delphi 2 para suportar a automação OLE. São variáveis de tipo indefinido, que em teoria podem receber qualquer valor. São utilizados no *DataSnap* para prover o *late-binding* com o *AppServer*, ou seja, a ligação tardia, onde o compilador só irá identificar os métodos no momento da chamada.

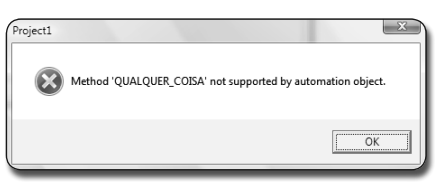



**Figura 21**. Late-binding pode gerar erros em runtime



Clube101.indb 37 10.11.08 15:55:01

**Listagem 6.** Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
 Label1.Caption := IntToStr(
   DM.DCOMConnection1.AppServer.QUALQUER_COISA
     (DM.cdsDepartmentDEPT_NO.AsInteger));
end;
```

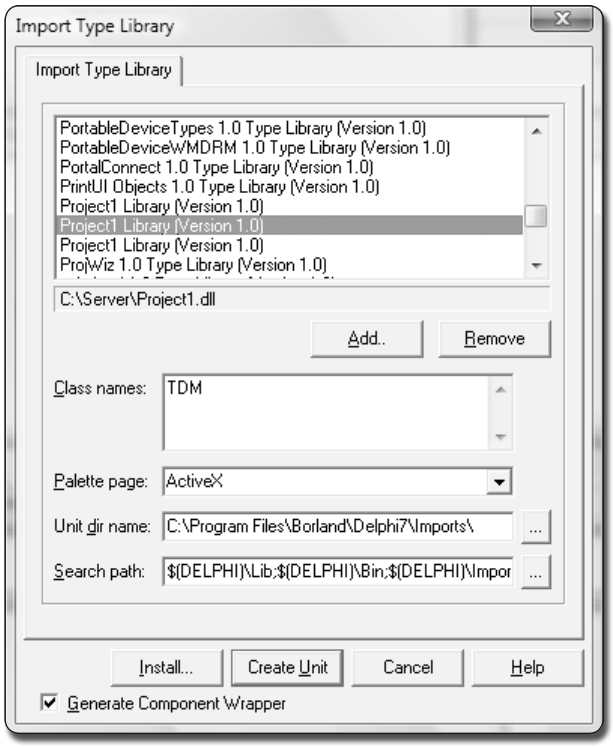



**Figura 22**. Importando a Type Library

**Figura 23**. Componente Wrapper

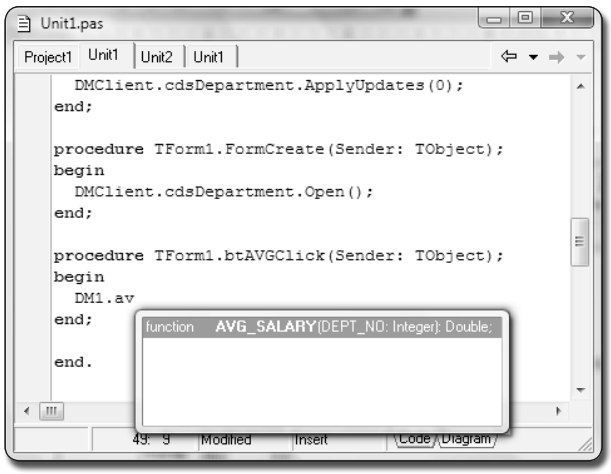



**Figura 24**. Early-Binding em ação

é que não temos suporte ao *Code Insight* ao chamar os métodos de *AppServer*.

## Early Binding

Uma alternativa ao utilizar *late-binding* é usar *early-binding*, que faz uma ligação prévia com o servidor de aplicação e com isso o compilador é capaz de identificar os métodos em tempo de compilação, com suporte ao *Code Insight*. Isso é feito através da importação da *Type Library*. Na prática, isso é 5 vezes mais rápido do que usar *late-binding*.

No cliente clique em *Project>Import Type Libray* (**Figura 22**). Deixe marcada a opção *Generate Component Wrapper*, o que fará com que o IDE crie um componente *Proxy* que encapsulará a *Type Library* para facilitar a comunicação com o *AppServer*. Clique em *Install, Yes* e *Ok*. Um componente (**Figura 23**) surgirá na paleta *ActiveX*.

> **Nota**: Renomeie neste momento o DM cliente para *DMClient* para não conflitar com o DM da biblioteca de tipos do componente servidor. Troque também as referências no código (eventos do botão *Apply* e *Form Create*).

Coloque no *form* o componente criado, que deverá se chamar *DM1*, representando o servidor de aplicação remoto. No botão que calcula a média, veja que agora podemos usar o *CodeInsight* para chamar a biblioteca de tipos (**Figura 24**). Digite o código da **Listagem 7** e faça o teste.

## DataRequest

Uma outra alternativa a utilizar *Early-Binding* ou *Late-Binding* é utilizar a própria interface *IAppServer*, sem modificar a biblioteca de tipos. Isso pode economizar tempo, além de um novo registro da TLB caso o objeto já esteja instalado.

Para a comunicação de dados que não sejam necessariamente relacionados a *DataSets*, *IAppServer* oferece o método *DataRequest*, que na prática significa que você pode solicitar qualquer dado do *AppServer* e receber qualquer coisa como retorno.

No *AppServer*, no evento *OnDataRequest* do *DataSetProvider*, digite o código da **Listagem 8.** Note que o evento recebe



Clube101.indb 38 10.11.08 15:55:30

como parâmetro um *Input*, que pode ser algum comando vindo do cliente (que nesse caso vai ser o código do departamento). Recompile.

No cliente, no *OnClick* do botão AVG, chame o método *DataRequest* do *ClientDataSet* como mostra a **Listagem 9.** Execute a faça o teste. Note que não foi preciso modificar nem usar a biblioteca de tipos, pois tudo já é feito por *IAppServer*.

## Executando como biblioteca

Imagine a seguinte situação. Você criou um poderoso sistema 3 camadas, com vários objetos COM+ e vários clientes, porém um usuário mais humilde resolveu comprar sua solução e vai rodá-la inteira numa simples máquina (nesse caso Banco de Dados + *AppServer* + cliente residiriam na mesma máquina).

Nessa situação, não haveria a necessidade de uma solução multicamadas, você poderia optar por usar *DataSnap* local. Mas graças ao COM+, temos uma elegante saída que pode otimizar sua aplicação para esse cenário.

Tal solução consiste em rodar o servidor como uma *biblioteca*, e não como um servidor. Nesse caso, quando o cliente ativar a comunicação com o *AppServer*, ao invés de ser levantado mais um processo *DLLHost.exe* para hospedar a DLL e conseqüentemente estabelecer um processo de comunicação *out-of-proc* (entre processos, envolvendo *marshalling* de dados), o servidor inteiro (DLL) é levantado no mesmo endereço de memória do processo chamador. Na prática, isso é como se servidor e cliente fossem compilados em um único EXE, pois a comunicação é *in-proc* (no mesmo processo), sem perda de desempenho e *marshalling*.

Para ver isso na prática, no gerenciador do COM+ dê um clique de direita no nosso aplicativo *ClubeDelphi*. Em *Activation*, marque a opção *Library Application* (lembre-se de voltar ao *default* – *server* – após o teste), como mostra a **Figura 25**. Agora execute o cliente. Em *View>Debug Windows>Modules*, note que são mostradas todas as DLLs que foram carregadas no mesmo endereço do processo EXE cliente, incluindo o próprio servidor *DataSnap* (**Figura 26**). Acredito que com isso você não tenha mais motivos para não migrar para *DataSnap* sua solução *client/server*.

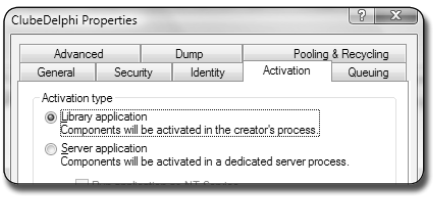



**Figura 25**. Opção Library Application

### Nota do DevMan

Muito cuidado ao modificar interfaces COM. Como comentado anteriormente, interfaces são imutáveis, então planeje bem seu servidor de aplicação antes de implementá-lo. Normalmente uma mudança nos métodos do servidor requer um novo registro da biblioteca e uma nova geração da *Type Library*.

### Nota do DevMan

A solução com *Library* torna o aplicativo duas camadas. Uma opção adicional seria rodar o *Firebird* como *Embedded*, aí o próprio BD rodaria no mesmo processo do cliente onde já está o servidor de aplicação, tudo em uma camada só. Para mais informações veja a minha aula em *http://www.devmedia.com.br/articles/viewcomp.asp?comp=1734*

**Listagem 7.** Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
  Label1.Caption:= FloatToStr(DM1.AVG_SALARY(
    DMClient.cdsDepartmentDEPT_NO.AsInteger));
end;
```

**Listagem 8.** OnDataRequest do DataSetProvider

```
function TDM.dspDepartmentDataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
begin
  SQLStoredProc1.ParamByName('COD_DEPT').AsInteger := Input;
  SQLStoredProc1.ExecProc;
  result := SQLStoredProc1.ParamByName('TOTAL').AsFloat;
end;
```

**Listagem 9.** DataRequest

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
    Label1.Caption := DM.cdsDepartment.DataRequest(
       DM.cdsDepartmentDEPT_NO.AsInteger)
end;
```

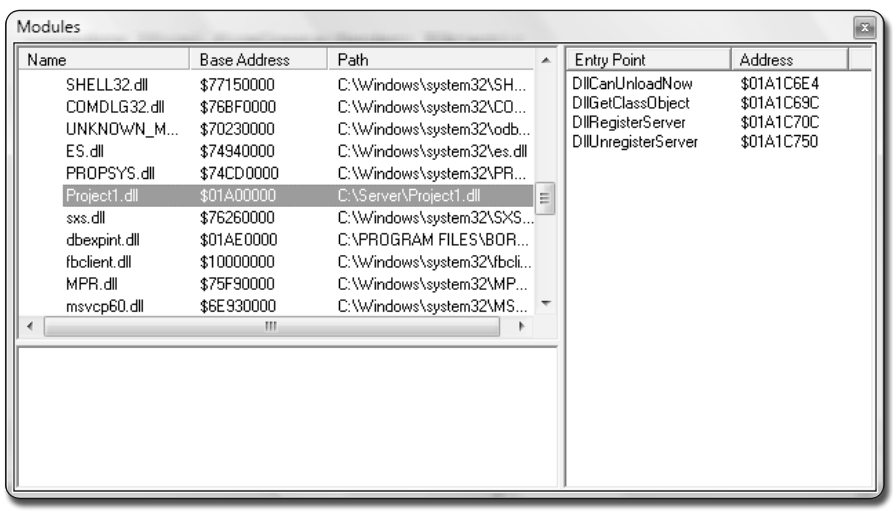



**Figura 26**. Servidor DataSnap inteiro rodando junto ao cliente



Clube101.indb 39 10.11.08 15:55:32

## Debugando servidores

Um procedimento comum no desenvolvimento *DataSnap* consiste em debugar o servidor de aplicação. Mas como debugar se ativação é *Just-in-time* (na hora da chamada cliente)? Ou seja, não podemos executar o servidor pois é uma DLL, precisamos executar o cliente e debugar a DLL. Para isso, alguns passos são necessários. No servidor, clique em *Project>Options*. Na aba *Linker* marque as opções conforme a **Figura 27** (lembre-se de voltar ao padrão após o *debug* para não afetar a otimização da DLL). Na aba *Compiler* configure como mostra a **Figura 28.**

Agora, em *Run>Parameters*, configure como mostrado na **Figura 29**. Note que colocamos o *Host Application* como o processo *DLLHost*. Em *parameters*, colocamos o seguinte:

```
/Processid:{1DC8FE61-0CEB-498D-823A-C3C3C4169C7F}
```

Esse GUID é o do pacote servidor, e com certeza vai mudar na sua máquina. Então pegue-o no catálogo do COM+, dando um clique de direita no nome da aplicação e selecionando o *Application ID*. Execute o cliente e clique no botão AVG. A **Figura 30** mostra o exemplo sendo depurado.

## Parâmetros

E finalmente, para concluir o exemplo, não poderíamos deixar de falar em parâmetros. No presente momento, estamos trazendo todos os departamentos e para cada um, todos os seus respectivos funcionários. Isso, sem dúvida, é muita informação para ser trafegada em um ambiente multicamadas. Que tal pedir ao cliente para que digite o código do departamento, e aí sim trazemos somente o departamento solicitado com seus respectivos funcionários?

O primeiro passo é trocar a instrução SQL do *qryDepartment* para:

```
select * from DEPARTMENT
where DEPT_NO=:DEPT_NO
```

Configure o parâmetro criado como sendo do tipo *Input* e *Integer*. Recompile o servidor de aplicação.

No cliente, coloque um *Edit* e um *Button* como mostra a **Figura 31.** No *ClientDataSet*, dê um clique de direita e escolha a opção *Fetch Params*, o que fará com que o parâmetro definido no *AppServer* venha parar na aplicação cliente e possa ser passado diretamente no *ClientDataSet* (**Figura 32**). O resto é por conta de *IAppServer*.

Retire o *Open* do *OnCreate* do *form* e no *OnClick* do novo botão digite o código da **Listagem 10**. Execute e digite um código de departamento no *Edit* (ex.: 600) e clique no botão. Veja que somente o departamento desejado é trafegado na rede (**Figura 33**), junto com seus respectivos funcionários através do *DataSetField*.

## Deploy

Até agora, utilizamos a mesma máquina para ambos cliente e servidor, apesar de serem dois processos. Mas como fazer isso funcionar em uma *intranet* real, com duas ou mais máquinas físicas? Pois bem, aproveitei minha infra-estrutura de rede em meu escritório para realizar os testes. Ao executar a aplicação em outro terminal, obtivemos um erro, claro, pois o cliente não sabe onde está o servidor e nem como acessá-lo ("Classe não registrada"). Mas o procedimento é simples:

1 - No servidor, compartilhe o disco ou o diretório onde está a aplicação servidora (*Server*);

2 - Na máquina cliente, por compartilhamento de rede, mapeie o diretório/disco compartilhado no passo 1;

3 - Na máquina cliente, digite o seguinte no *prompt* para registrar a *Type Library*:

```
tregsvr z:\Server\Project1.dll
```

**Listagem 10.** Passando parâmetros via IAppServer

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DMClient.cdsDepartment.Close();
  DMClient.cdsDepartment.Params[0].AsInteger :=
    StrToInt(Edit1.Text);
  DMClient.cdsDepartment.Open();
end;
```

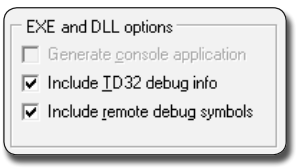



**Figura 27.** Opções para Debug

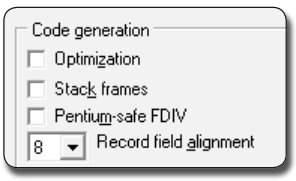



**Figura 28.** Mais opções para Debug

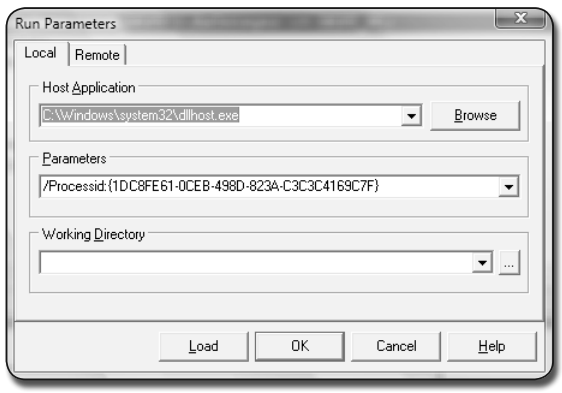



**Figura 29.** Run Parameters

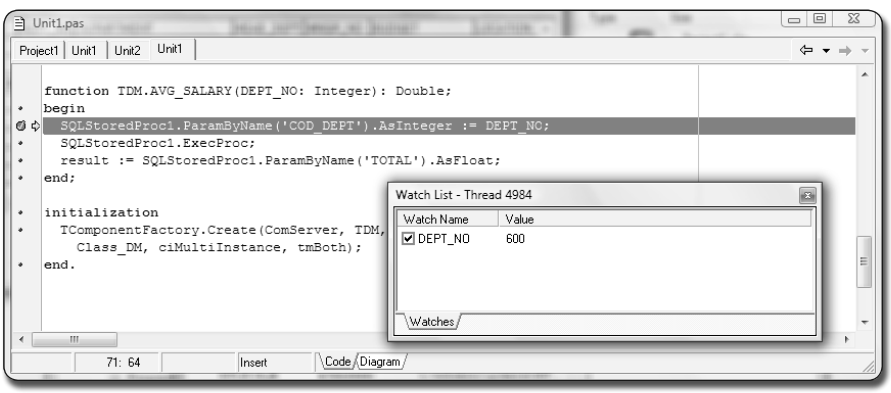



**Figura 30.** Depurando a aplicação servidora



Clube101.indb 40 10.11.08 15:55:34

Onde *z* é o disco onde está aplicação servidora e o restante o caminho da DLL. *TRegSvr* é um aplicativo que vem com o *Delphi* e pode ser copiado para o local onde você vai executar o comando no *prompt*.

Pronto! Temos o cliente acessando o servidor, através de uma LAN, e a chamada remota sendo executada perfeitamente.

> **Nota**: Se não existir a biblioteca MIDAS.DLL no cliente, copie-a para essa estação (diretório *System* do *Windows*), ou declare *MidasLib* em uma unit qualquer da aplicação cliente, recompile e refaça o *deploy*.

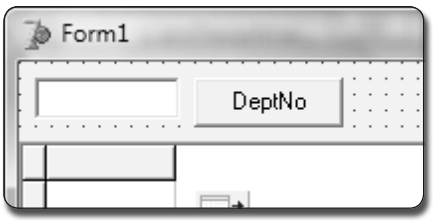



**Figura 31**. Formulário pronto para passagem de parâmetro

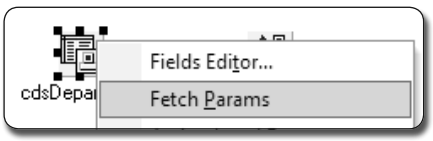



**Figura 32**. Trazendo os parâmetros via IAppServer

## Conclusão

O desenvolvimento *DataSnap* é elegante, rápido, produtivo e robusto. Sabendo aplicar bem as técnicas aqui demonstradas, vimos como criar uma solução escalável para um ambiente multicamadas. Acredito que não existam mais motivos para você não migrar mais sua solução *client/server* para um próximo nível em boas práticas de desenvolvimento. ●

**Links**

*http://www.distribucon.com/*

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

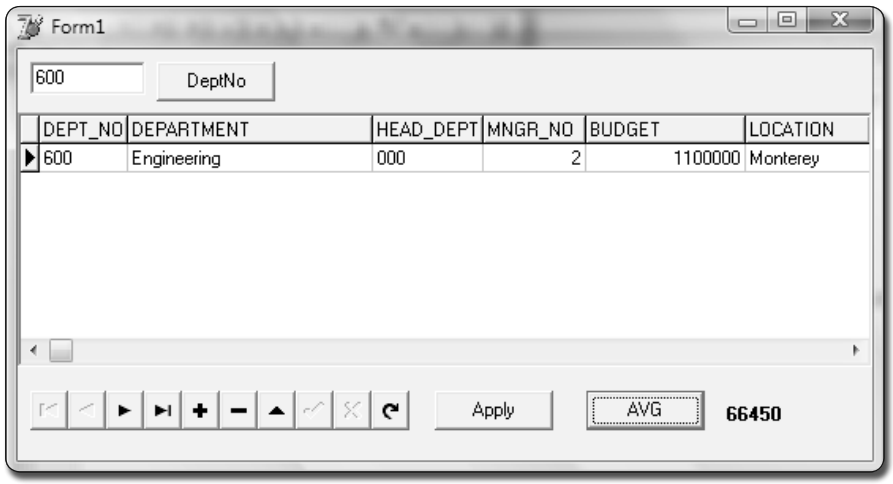
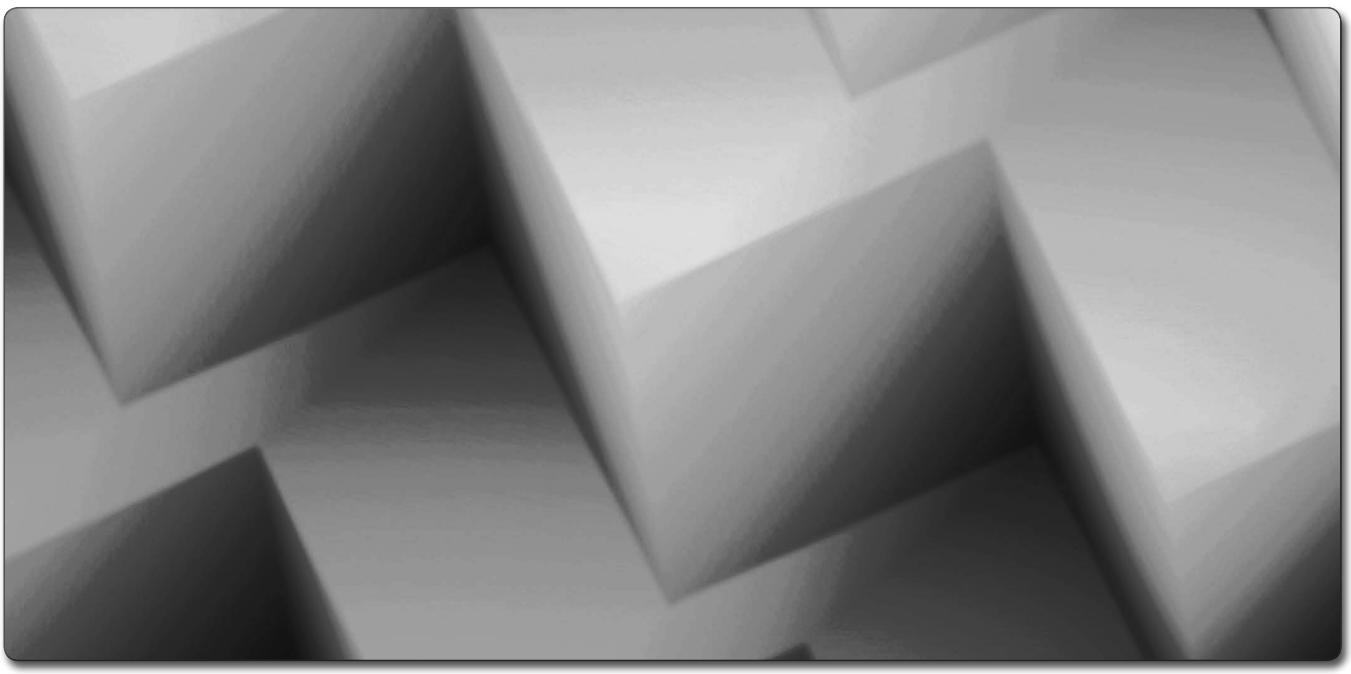



**Figura 33**. Opção Library Application





Clube101.indb 41 10.11.08 15:56:08