Preparing For A Terminal Server Environment

by Jani Järvinen

If you have been following the technological talks during the past few years, you have probably heard of Windows Terminal Services. This allows you to convert a Windows NT or Windows 2000 system to a multi-user server capable of running multiple desktops at the same time. In this article, I will introduce you to the key concepts and help you in programming for this environment.

From time to time, history seems to repeat itself. Many years ago, applications were running on the server, and the client computers were considered to be terminals. Such a terminal didn't have much processing power on its own, but it had a screen, a keyboard and a network connection (or a simple RS232 serial cable).

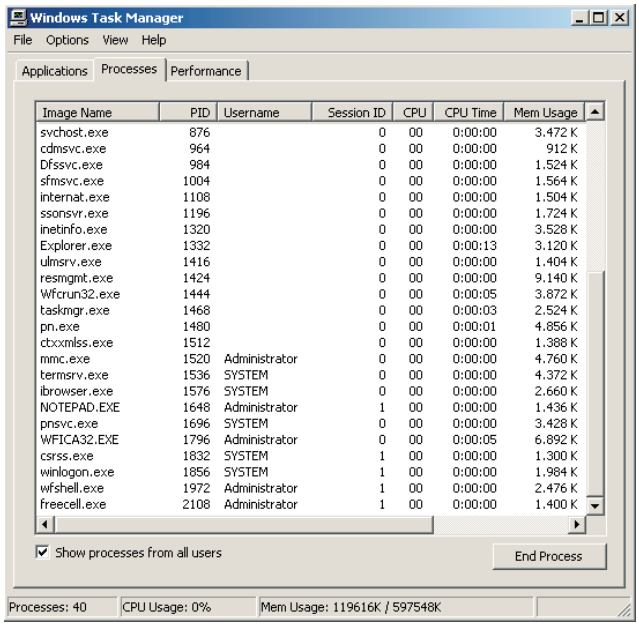
Today, the personal computer landscape is vastly different. Even the average PC sitting on the office desk is powerful enough to process video, handle audio and display rich user interfaces. Processing power allows us to manipulate lots of data at once, and fast peripherals handle swift communications with the outside world.

From the user's and developer's perspectives, personal computers can be fun, since they are fast, flexible and entertaining. But, from the management point of view, PCs can easily create many problems. Different hardware and software configurations can become a maintenance nightmare. Even if hardware is cheap and software close to free, the cost of supporting a regular PC, and its user suffering from the Monday morning syndrome, can simply be too high.

To help reduce the total cost of ownership (TCO), people like Oracle's Larry Ellison have proposed network PCs: simple terminal-like devices that would be able to run web and Java applications. Unfortunately, these devices haven't become commonplace, at least not yet. Although developing web applications with Delphi is straightforward (as you have seen on the pages of this magazine), converting existing applications to the web is often simply too slow.

Enter Terminal Services

Although web-based applications and network computers can be efficient solutions to certain types of problem, they do not necessarily fit into a regular Windows environment: as we all know, Windows applications are still number one on the corporate desktop.



➤ *Figure 1: Installing Terminal Services allows multiple users to run their applications.*

The idea of Windows Terminal Services is to move processing from the client desktop to the server, while allowing the user to interact with the application just as if it were a normal client application. The Terminal Services technology transfers mouse and keyboard input from the client to the server, and screen images from the server back to the client.

When I was first introduced to the concept, my immediate reaction was that Terminal Services was just another piece of software that allows remote administration, like pcAnywhere, Reach Out or NetOp. Compared to Terminal Services, the idea of these tools is similar, but the implementation is completely different.

Conventional remote control software hooks up to the user's desktop and allows control of the remote computer using the local keyboard and mouse. On the other hand, Terminal Services allows you to run multiple interactive Windows sessions on the same computer. For example, both John and Jane could have an interactive session running on the same system. And just like remote control software, both John and Jane can be anywhere in the world and still be able to run the applications they need.

Technically, a Terminal Services enabled computer can also have a regular console session in addition to the remote sessions for John and Jane. However, most commonly a Terminal Services enabled computer runs on the corporate computing centre, without even a monitor connected, since all operations, including administration, can be done over the wire.

Before a user can connect to a Terminal Server computer remotely, he or she needs a small client application. The purpose of

this client application is to capture mouse and keyboard activity on the client and send the screen images from the server to the client. Because of this operation model, it doesn't actually matter what operating system the client runs. For instance, a user running Macintosh or Linux could easily hook up to a Windows 2000 server and run the applications he or she needs.

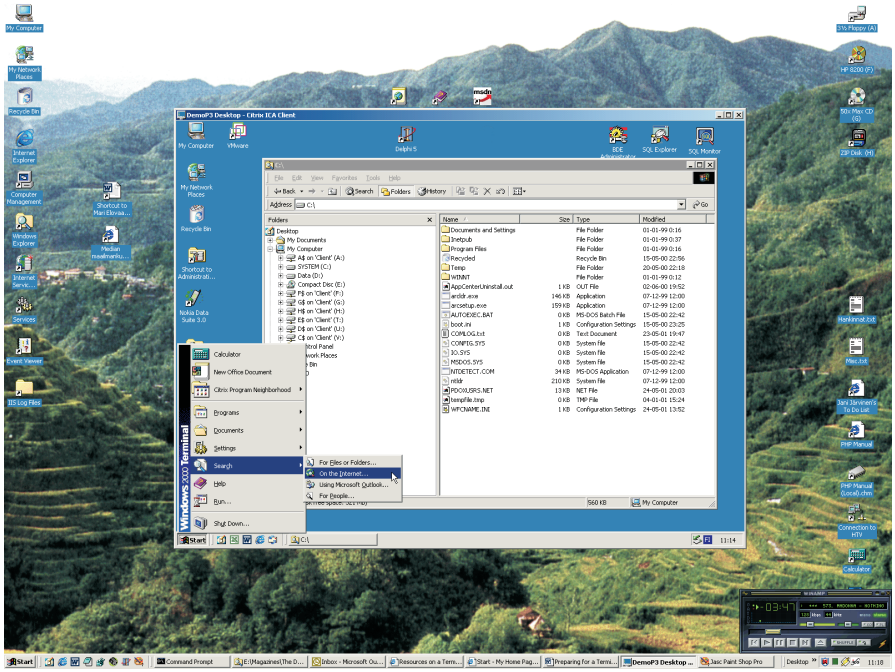
The previously mentioned ability to mix and match operating systems isn't the only benefit of Terminal Services. Other key benefits include the ability to run heavy applications even on an old 486 client (since all the processing happens on the server) and the support for multiple connection methods (from modem to direct LAN connection). Plus, an application needs to be installed only once on the server to become available to all client computers.

Background Information

At this point you are probably interested in knowing how Terminal Services works. Traditional Windows NT and Windows 2000 systems are single-user operating systems, meaning that only one user can execute applications on the system. When Terminal Services is enabled on the system, multiple users can connect to the server and share its memory, CPU, disk and other resources.

When talking about Terminal Services, it is important to know the players. The US-based company Citrix, Inc is the father of a Windows-based Terminal Services solution. Citrix's current product family is named MetaFrame, the newest version being MetaFrame XP, short for MetaFrame eXtended Platform (as opposed to eXPerience in Windows XP).

Citrix was the first vendor to develop a multi-user Windows kernel (MultiWin, see Figure 1), and as Microsoft noted the success of the Citrix products, it released Windows NT 4.0 Terminal Server Edition. The Microsoft implementation is based on Citrix's work, and, for example, the management tools look quite similar on both



➤ *Figure 2: A desktop on a desktop using Citrix MetaFrame.*

products. As you would guess, Citrix provides additional features and, for example, supports many other client operating systems than just Windows systems.

With the introduction of Windows 2000, Microsoft decided to put Terminal Server features into Windows 2000 Server without additional cost. This move has greatly increased the demand for applications compatible with Terminal Services. Note that in this article, I'm using the term *Terminal Services* to mean both Microsoft and Citrix solutions, both on Windows NT 4.0 and Windows 2000. If I need to make a distinction, I'll do so explicitly. TSE is a common abbreviation of Terminal Services Environment.

From a technical point of view, installing Terminal Services on a system modifies the internal kernel components so that they become aware of multiple users. Each connection to the Terminal Services creates a new *session*. You could consider this session exactly identical to a normal *console session* except that the desktop of the user is not shown on the screen of the server, but instead on the client.

When a user connects to a Terminal Server, the system creates a session for the user. When the user launches an application, the forms and graphics the application displays are drawn to an internal screen buffer. Whenever this buffer changes, the changes are transmitted over the network to the client.

Since the application is running on the server, it is very important to understand that applications share the resources on the server. For example, RAM, disk, network and CPU resources are shared by all sessions and by all applications running in them. For instance, if one application suddenly starts to consume 100% of the CPU time, there isn't much left for other applications.

Jumping to the client side, it is essential to know how the remote application interacts with the applications running natively on the client. Of course, not all client computers might be able to run applications on their own (they might not even have a hard disk), but most commonly, a Terminal Services client computer is simply a Windows-based PC.

Frequently, when a user initiates a connection to a Terminal Server system, he or she is presented with a desktop inside a window. After logging on, the Terminal Server system loads the desktop of the user based on the user name. The results look like Figure 2. That is,

the user sees a desktop in a desktop, so to say.

For the experienced user, this notion of showing a desktop on a desktop is great, but the average user can become confused with two almost identical `Start` menus and file systems. Of course, the remote desktop can be set to occupy the whole screen, but things can still get complicated.

To solve the problem, Citrix has developed a feature called seamless window integration, in which an application (say, Notepad) runs on the client desktop just like it would have been started as a normal client application. With this feature, the remote desktop is not shown, leaving only the Notepad window visible. This feature is not available with the Microsoft solution.

Application Design Considerations

So far, I've only discussed how Terminal Services works. As a software developer, you must also learn how Terminal Services affects your applications. The good news is that most applications work inside a Terminal Services Environment 'as is'. However, if you want your application to behave nicely in such situations, there are many points to consider.

Since I only have limited space available, I suggest that you also read the documentation available in the Microsoft Platform SDK, viewable at http://msdn.microsoft.com. This documentation contains detailed information on the subject: I will concentrate on key points.

First and foremost, you should try to conserve resources when your application is running in a Terminal Server session (I will soon show you how to detect this). Since everything happening on the screen must be transferred from the server to the client, you should minimize the action occurring on your forms. For example, if you are using graphical feedback, such as AVI videos, during long operations, it might be better to come up with a less intensive graphics effect. Similarly, you should disable splash screens and other entertaining features. Even the blinking of a text caret causes network traffic!

In addition to graphics, you should also minimize your requirement for sound effects. Unless your application is an audio manipulating application, it is best to simply disable all audio effects in your application if it is running inside a Terminal Server session. Leaving audio on in your application does not cause the remote server to emit the sounds, since the Terminal Services solution is intelligent enough to transfer the audio data to the client. Since audio data consumes lots of bits, all this comes down to saving network bandwidth. Also, audio might simply be disabled on the client without your application knowing it.

Basic features such as clipboard access should also be considered. Your application running on the Terminal Server shares the clipboard of the client computer, as opposed to the remote computer. This means that all clipboard operations must be transferred over the wire, so you should avoid copying lots of data to and from the clipboard.

Note that clipboard integration is very important for the end-user. For example, John could run an instance of Notepad locally on his computer and Word remotely. It is easy for him to copy and paste information between these two applications, since nothing special needs to be done.

➤ *Listing 1: Retrieving the value of an environment variable.*

```
Function GetEnvironmentVariable(Name : String) : String;
Var I : Integer;
Begin
  Result := '(cannot determine)';
  I := Windows.GetEnvironmentVariable(PChar(Name),nil,0);
  If (I > 0) Then Begin
    SetLength(Result,I-1);
    Windows.GetEnvironmentVariable(PChar(Name),PChar(Result),I);
  End;
end;
```

How printing is implemented is another important issue to understand. Since applications running inside Terminal Services try to emulate regular client applications as much as possible, a feature called client printing is supported. Client printing means, effectively, that a remote application is able to print to a printer attached directly to the client computer.

If your application does lots of printing, you might wish to optimize printing, since every printing command must be transferred from the server to the client if printing to a client printer. If printing to a printer attached to the Terminal Server, the network is not utilized.

Just like printers, Terminal Services also supports serial devices attached to the client computer. For example, if Jane is using a digitizing board attached to her client computer, the remote application should be able to use it normally. However, some serial devices are special and require tweaking and tuning to make them work in a TSE.

Files, Registry And Kernel Object Names

Access to files and the registry requires thought as well. Since the file system on the Terminal Server is shared by all applications running on it, you must be cautious about not overwriting data that other instances of your application might be using. For instance, if your application has the habit of using a temporary file with the same name every time, you might run into problems when multiple instances of your applications are running on the same system.

To help ensure applications are robust, a Terminal Server session always redirects the `TEMP` environment variable to point to a unique directory. For example, if `TEMP` usually points to C:\TEMP, Terminal Services automatically modifies it to point to C:\TEMP\1, C:\TEMP\2 and so on depending on the session number. So, if you are using temporary files, it is good practice to use the `TEMP` environment variable to get the path to the temporary directory. The helper function

in Listing 1 allows you to easily get the contents of any environment variable.

Other than files, registry access is the Achilles heel of many applications. If your application stores information in the HKEY_CURRENT_USER hive, the settings are only available to a single user. If another user logs on to the system with other credentials, Terminal Services switches the contents of the HKEY_CURRENT_USER hive to match the settings of the other user. If your application relies on finding key information in this hive, your application might not work in the Terminal Services environment.

To remedy the situation, it is best to store global computer settings (such as pathnames) in HKEY_LOCAL_MACHINE and only user-related settings (such as window positions) in HKEY_CURRENT_USER. Note that this is a really common mistake to make: even Borland has neglected this with Delphi. Delphi 6 fixes this problem, however.

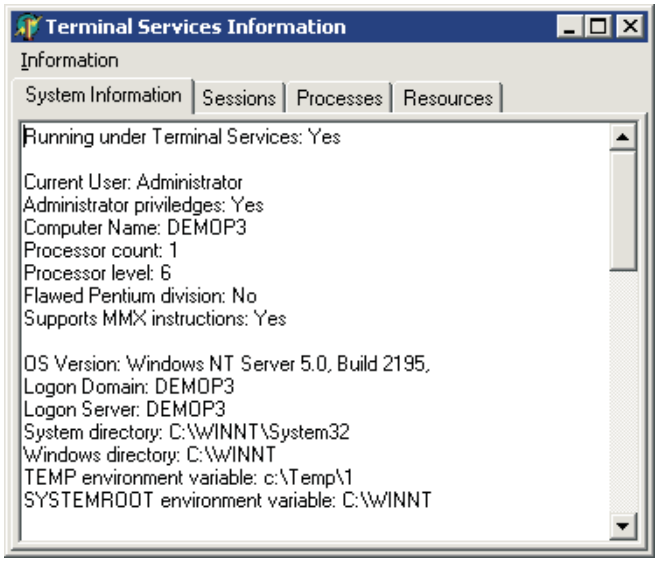
For advanced applications, Win32 kernel object names can also be an issue. When an application is running regularly on the client without the presence of Terminal Services, kernel object names share one global namespace. So, it is easy create a named kernel object, for instance an event, and make sure all instances of the application share the same event object. Or the presence of such an object could mean that an instance of your application is already running. It would then be easy to terminate the second instance, forcing the user to have only a single instance of your application.

Inside a Terminal Services session, things are different. Each session has its own private kernel object namespace. I mean that if your application running under John's session creates a named kernel object, another instance of your application running inside Jane's session cannot see the object.

In NT 4.0, there wasn't much you could do to solve the problem. On the other hand, Windows 2000 provides a neat solution to the problem by introducing global and local namespaces. By default, an application running inside a Terminal Services session uses the local namespace. Thus, it can only see the objects created inside that session.

However, by prefixing a kernel object name with Global\, the application can specify that the global namespace should be used. Similarly, the Local\ prefix can be used to specify the local namespace. If your application is running on a Windows 2000 system without Terminal Services, the prefix is ignored. Remember to be careful with older Windows versions, since the backslash character is invalid in NT 4.0, Windows 9x and Me.

➤ *Figure 3: The sample application is able to display system information.*



Terminal Services APIs

If you want to fine-tune your application for Terminal Services, or you wish to develop applications to manage Terminal Services, you need to use the Terminal Services APIs. Most of these API functions are implemented in WTSAPI32.DLL, and they are prefixed with the letters WTS, for example WTSEnumerateSessions.

Using these APIs is simple, although the header files that ship with Delphi don't contain the necessary header translations. Instead, you must either manually translate the headers from C code or obtain the translations ready-made. The JEDI Project (www.delphi-jedi.org) has done such a translation, but I've also translated several important API functions for the purposes of the example application. To do the translations yourself, you need to obtain the header file WTSAPI32.H, among other files. It is easiest to get these by downloading the Platform SDK package from Microsoft.

The WTS functions constantly refer to server and session handles. These handles are used to identify a server running Terminal Services and a session inside that server. To specify the current server, you can always use the constant WTS_CURRENT_SERVER_HANDLE, defined as zero in WTSAPI32.H. Similarly, the current session can be referred to with the WTS_ CURRENT_SESSION constant.

The example application I wrote for this article is named Terminal Services Information. The purpose of this application is to retrieve system information about the computer on which the application is running, and also to demonstrate the use of the WTS APIs. The user interface of the application is divided into four tabs, as shown in Figure 3. Note that to run the sample application, you will need at least Windows NT 4.0 Terminal Server Edition or Windows 2000 Server. The application won't start in other environments due to missing kernel features.

The first tab, System Information, displays miscellaneous details about the current computer. For example, this includes memory information, networking settings and printer information. I won't be going through all of the code since it would require at least one more article! However, I would like to draw your attention to the first information displayed by the

application. The code is shown in Listing 2.

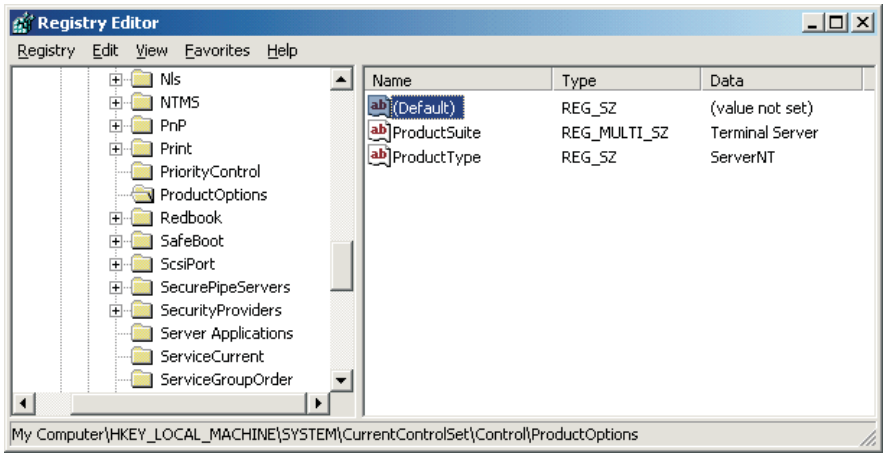
As you can see, the code first checks to see if the current operating system is Windows 2000 (or later) and, if so, calls the `GetSystemMetrics` API function. The parameter for this function is the constant `SM_REMOTESESSION`. It is not defined in the source files that come with Delphi, but you can always use the following translation (from WinUser.h):

```
Const
  SM_REMOTESESSION = $1000;
```

If the return value from `GetSystemMetrics` is non-zero, the result means that the current application is running inside a Terminal Server session. If the return value is zero, the application is being run using conventional methods. Note that this test only works with Windows 2000. With NT 4.0, there isn't a documented method for testing if the application runs inside a terminal session.

However, if you are using NT 4.0, you can look at the registry keys under this location:

```
HKEY_LOCAL_MACHINE\SYSTEM\
  CurrentControlSet\
  Control\
  Product Options
```



➤ *Figure 4: Trying to detect Windows NT 4.0 TSE requires checking the registry.*

(see Figure 4). If the `ProductSuite` key is set to `Terminal Server`, it is quite likely that your application is in fact running inside a terminal session.

Getting Session Information

The second tab of the sample application shows a screen like the one in Figure 5. With the buttons on this tab, you are able to retrieve session information given a process ID and also list the sessions on the current server. When a process is running, it has its own unique process ID, which is guaranteed to be unique while the process is running. Similarly, each terminal session has a session ID. To determine in which session a process is running given its process ID, you can call the `ProcessIdToSessionId` API function. It is defined as:

```
Function ProcessIdToSessionId(
  ProcessID : Integer;
  { from WinBase.h }
  Var SessionID : Integer) :
  Bool; StdCall;
  External Kernel32
  Name 'ProcessIdToSessionId';
```

To fill in the list of active sessions, the example application uses the API function `WTSEnumerateSessions`. This function can only be called if Terminal Services has been installed, or when running inside a terminal session. Otherwise the function fails and `GetLastError` returns `ERROR_APP_WRONG_OS`. This is common to other WTS API functions as well.

If the call to `WTSEnumerateSessions` is successful, the function returns a pointer to an array of structures that can be used to identify the sessions. The structure itself identifies three things: the session ID, the window station name, and finally the session state

➤ *Listing 2: Detecting if running inside a Terminal Services session.*

```
Const
  BoolArray : Array[False..True] of String = ('No','Yes');
Function GetTerminalServicesInfo : String;
Var B : Boolean;
Begin
  Result := 'Running under Terminal Services: ';
  If ((Win32Platform = VER_PLATFORM_WIN32_NT) And
      (Win32MajorVersion >= 5)) Then Begin { Windows 2000 or later }
    B := (GetSystemMetrics(SM_REMOTESESSION) <> 0);
    Result := Result+BoolArray[B];
  End Else
    Result := Result+'(cannot determine)';
End;
```

➤ *Listing 3: Retrieving session information with WTSEnumerateSessions.*

```
procedure TTSIMainForm.SessionRefreshTimerTimer(
  Sender: TObject);
Var
  S   : String;
  P   : PSessionInfoArray;
  I,J : Integer;
  TI  : Pointer;
begin
  If (Not RefreshSessions.Checked) Then Exit;
  { update session states }
  If (Not WTSEnumerateSessions(WTS_CURRENT_SERVER_HANDLE,
    0,1,P,I)) Then Begin
    SessionRefreshTimer.Enabled := False;
    PageControl1.ActivePageIndex := 1;
    TSSessions.SetFocus;
    RaiseLastWin32Error;
  End;
  TSSessions.Items.Clear;
  For J := 0 to I-1 do Begin
  {$R-}
    TI := TypeInfo(TConnectState);
    S := GetEnumName(TI,Integer(P^[J].State));
    With TSSessions.Items.Add do Begin
      Caption := IntToStr(P^[J].SessionID);
      SubItems.Add(P^[J].WindowStation);
      SubItems.Add(S);
    End;
  {$R+}
  End;
  WTSFreeMemory(P);
end;
```

information. The window station is the internal name of the (invisible) USER environment containing the desktop and the clipboard, among other things. Session state is an enumeration: it specifies whether the session is currently active, waiting for a connection, or initializing, and so on.

The array returned by WTSEnumerateSessions is a variable-sized array, so you need to be careful about the compiler range checks when accessing the array. For example, I personally always keep range checks on ({$R+}) during development, because this allows me to catch annoying bugs faster. There's no excuse for not using $R+ since it is very easy to enable and disable range checks if needed, as you can see from Listing 3.

Once you have finished with the array returned by WTSEnumerateSessions, use the WTSFreeMemory API function to free the memory buffer. Do not use the memory management functions that Delphi provides to free the buffer.

The example application also contains two buttons that can be used to interact with sessions. The first button, Show Session Information, is able to display more information about a session that can be retrieved with a call to WTSEnumerateSessions. The code uses the WTSQuerySessionInformation API function, which returns a lot of information given a server handle and a session ID. Just as with the WTSEnumerateSessions call, the buffer returned by WTSQuerySessionInformation must be freed with a call to WTSFreeMemory.

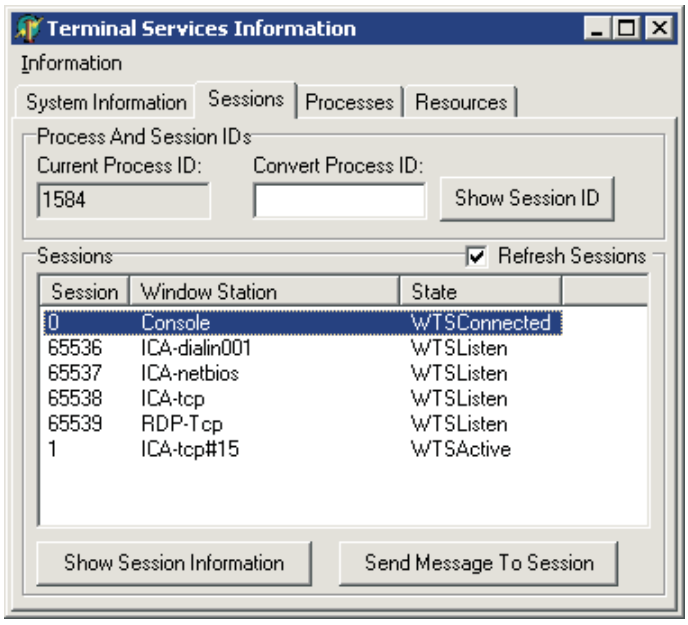
The second button executes the code in Listing 4. By clicking the button, you are able to send a message to the selected session. The given message then pops up on the console of the session. By calling WTSSendMessage, you are able to send quick notifications to users. For example, if you are building an administrative utility, you could easily enumerate the available sessions and send a message to each stating that the system is going to be rebooted in 10 minutes. Neat, I'd say.

```
procedure TTSIMainForm.SendMsgToSessionClick(Sender: TObject);
Var
  Session  : LongWord;
  Title    : String;
  AMessage : String;
  I        : Integer;
begin
  If (TSSessions.Selected = nil) Then Begin
    ShowMessage('Please select a session first.');
    Exit;
  End;
  Session := StrToInt(TSSessions.Selected.Caption);
  Title := 'Hello Session #'+IntToStr(Session);
  AMessage := 'It is now: '+DateTimeToStr(Now);
  If (Not WTSSendMessage(WTS_CURRENT_SERVER_HANDLE,Session,
    PChar(Title),Length(Title),PChar(AMessage),Length(AMessage),MB_OK,
    0,I,False)) Then RaiseLastWin32Error;
  ShowMessage('Message sent.');
end;
```

➤ *Listing 4: Send a message to a remote session.*

```
procedure TTSIMainForm.RefreshProcessesClick(Sender: TObject);
Var
  P   : PProcessInfoArray;
  I,J : Integer;
begin
  If (Not WTSEnumerateProcesses(WTS_CURRENT_SERVER_HANDLE,0,1,P,I)) Then
    RaiseLastWin32Error;
  ProcessList.Items.Clear;
  ProcessCount.Caption := IntToStr(I)+' processes shown';
  For J := 0 to I-1 do Begin
  {$R-}
    With ProcessList.Items.Add do Begin
      Caption := IntToStr(P^[J].ProcessID);
      SubItems.Add(IntToStr(P^[J].SessionID));
      If (P^[J].ProcessName = nil) Then SubItems.Add('-')
      Else SubItems.Add(P^[J].ProcessName);
      SubItems.Add(SIDToUserName(P^[J].UserSID));
    End;
  {$R+}
  End;
  WTSFreeMemory(P);
end;
```

➤ *Listing 5: Using WTSEnumerateProcesses.*

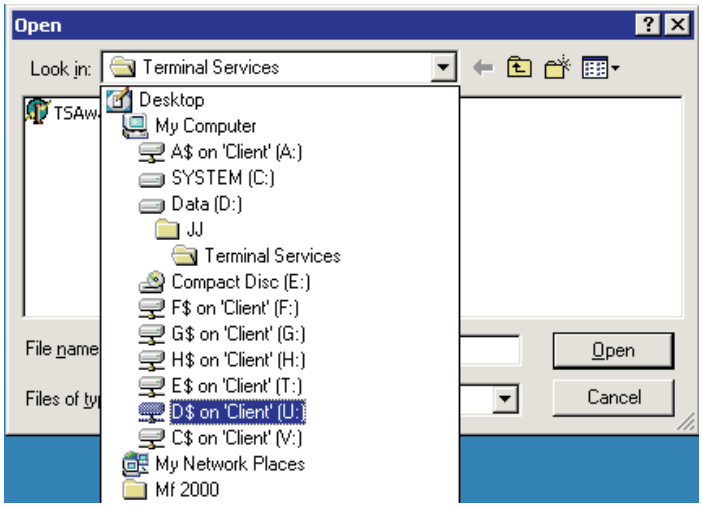


➤ *Figure 5: Displaying session information.*

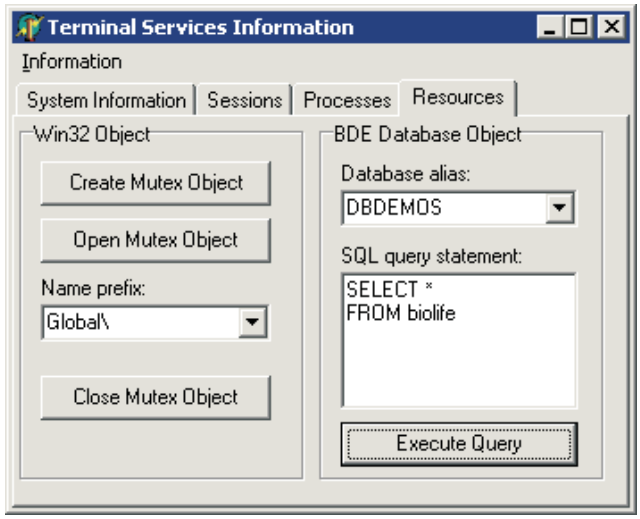
Listing Processes

Since a Terminal Server is able to serve multiple users at the same time, there can be potentially dozens of processes running in the system at any given moment. Being able to list these processes can be helpful if you are building administrative tools for Terminal Services or you want to know how many instances of your application are currently running, assuming that the name of your EXE is unique enough.

The WTS API also provides a function called WTSEnumerateProcesses which allows you to get a list of running processes. The code that demonstrates how to use this function is shown in Listing 5. Just as with WTSEnumerateSessions, WTSEnumerateProcesses returns a buffer that contains a variable-sized array of process information. Each running process is identified



➤ *Figure 6: The client's C: drive becomes a V: drive inside a remote session.*



➤ *Figure 7: Creating Win32 kernel objects and executing SQL queries.*

with the fields in the `WTS_PROCESS_INFO` structure. I've translated this into a record of type `TProcessInfo`.

The interesting thing about this record is that one of the fields specifies the security identifier (SID) of the user running the process. The security identifier is a way for Windows NT and Windows 2000 to identify a user. To convert an SID to a user name, you can use the `LookupAccountSid` API function, declared in WINDOWS.PAS.

To convert an SID to a user name string, I've written the helper function `SIDToUserName`. This function is shown in Listing 6. Note that a null (`nil`) SID means the special `SYSTEM` account.

Conclusions

Preparing your applications to run in a Terminal Services environment requires more thought than coding. Since most applications run without modifications quite nicely in such an environment, there is no need for costly reprogramming given that your application is 'well behaved'. Certain special applications, most probably those which are accessing hardware, are the ones that will require most work. Of course, I'm not saying that other kinds of applications would not require any modifications: you need to experiment and see what work is needed.

Both Citrix and Microsoft have done great work in making applications run as smoothly as possible inside a Terminal Server session. For example, access to drive letters works almost the same compared to an application running locally (see Figure 6).

Similarly, things like sound, the clipboard and printing work just the same as on a local computer. Database access, for example, can work without any modifications, as proved by the example application and the `Execute Query` button on the `Resources` tab (see Figure 7).

If you want to fine-tune your application for a Terminal Services environment, you can use the WTS API functions. I have demonstrated how to use these functions to enumerate sessions and processes, retrieve session information and send messages to sessions. As you saw in the code listings, using these functions is actually quite simple once you have the necessary C header translations available. Check out the source code for the example application, included on the companion disk, for more details.

If you are really keen on the WTS APIs, you have the possibility to drill down even further. For example, the virtual channel APIs allow you to directly control what is put on the wire when the client and server computers need to communicate. However, that is well worth a new article.

Until next time!

➤ *Listing 6: Converting an SID to a user name.*

```
Function SIDToUserName(SID : PSID) : String;
Var
  Name  : Array[0..256] of Char;
  NLen  : Cardinal;
  Dom   : Array[0..256] of Char;
  DLen  : Cardinal;
  SType : Cardinal;
begin
  If (SID = nil) Then Result := 'SYSTEM'
  Else Begin
    NLen := SizeOf(Name);
    DLen := SizeOf(Dom);
    If (Not LookupAccountSid(nil,SID,Name,NLen,Dom,DLen,SType)) Then
      Result := '(unknown)'
    Else Result := StrPas(Name);
  End;
end;
```

Jani Järvinen works as a technical support person for Borland products. He is a Microsoft Certified Professional (MCP) and a Citrix Certified Administrator (CCA). Email him at janij@dystopia.fi