Windows 2000 Terminal Services

Optimizing Applications for Windows 2000 Terminal Services and Windows NT Server 4.0, Terminal Server Edition

Operating System

White Paper

Abstract

Terminal Services is a configurable service incorporated into the Microsoft® Windows® 2000 Server operating system that delivers the Windows 2000 Professional desktop and 32-bit Windows-based applications to diverse desktop platforms. Terminal Services can run any well-behaved Windows-based application, but its multiuser nature tends to expose flaws and shortcuts in applications.

This paper provides guidelines for ensuring that applications run well under Windows 2000 Terminal Services. It also provides information on enhancing the user experience by tuning the application for the Terminal Services environment, and taking advantage of the capabilities Terminal Services provides.

Introduction

Terminal Services is the configurable service included in the Windows 2000 Server operating system that gives it the capability to run 32-bit Windows-based applications centrally from a server. This technology was introduced in Windows NT® Server, Terminal Server Edition, version 4.0, where it continues to deliver the base terminal emulation functionality that customers ask for today. In the Windows 2000 operating system, Terminal Services are fully integrated with the Windows 2000 Server kernel. Terminal Services emulator clients are available for many different desktop platforms (MS-DOS®, Windows, Macintosh, UNIX, and others.) Non-Windows-based desktops require third party add-on software.

Unlike the traditional client/server environment, when Terminal Services are enabled in Windows 2000 Server, all the application processing occurs on the server. The Terminal Services client performs no local processing of applications, it just displays the application output. The Terminal Services technology transmits only the application presentation—the graphical user interface (GUI)—to the client. Each user logs on and perceives his or her session only, which is transparently managed by the server operating system and is independent from any other client session.

From an application development perspective, one of the biggest benefits of Terminal Services is that well-behaved 16- or 32-bit Windows-based applications run, as is—no programming changes are required to run them under Terminal Services. However, this does not mean that all existing applications run equally well under Terminal Services. Understanding how to design applications that take advantage of the new capabilities of Windows 2000 Terminal Services is important. It's also important to understand how bad programming habits are magnified in the Terminal Services environment, and to recognize where specific programming practices for the multiuser environment can be applied.

Following these guidelines does not limit or compromise the ability of an application to function in the traditional Windows-based client/server environment—Terminal Services-optimized applications work well in both environments.

Note This white paper examines optimization techniques any application developer can use to ensure his or her Windows-based application runs well under Terminal Services. There is also support in the Windows 2000 Server operating system for a small set of Windows APIs available for Terminal Services. This white paper does not include information on those APIs. A separate white paper is available in Microsoft TechNet titled "Using and Understanding APIs for Terminal Server."

Terminal Services Extends Multiuser Capabilities of Windows 2000 Server

In addition to providing a means to serve Windows-based applications to terminals and other thin client devices, Terminal Services extends the multiuser capacity of the Windows 2000 Server and Windows NT Server 4.0 operating system family. Of course, Windows 2000 Server and Windows NT Server 4.0 are inherently multiuser-capable in the following ways:

- User profiles stored on the server allow numerous users to see their own desktop preferences and settings and run various applications when they log on. Application features and behavior can be controlled by user profiles.
- Security policies control file access rights and permissions for both local and remote client users.
- Operating system interfaces allow simultaneous sessions in which users safely access common files and databases stored on file servers throughout a distributed network.

The Terminal Services technology goes beyond the client/server multiuser services listed above, which are an integral part of the Windows 2000 Server and Windows NT Server 4.0 operating system family. The Terminal Services architecture allows users and applications to share hardware and software resources commonly found on a Windows 2000 Professional or Windows NT-based clients in the traditional two or three-tiered client/server architecture. These resources, which instead are shared on the server, include use of a central CPU, memory, and storage, as well as operating system resources such as the registry and other data structures. Developers of applications written for Windows-based desktops can use the information in this white paper optimize their applications to run under Terminal Services.

Comparing Host-Based and Windows 2000-Based Terminal Services

In some ways, Terminal Services are analogous to older, centralized host or mainframe, environments. In the centralized host architecture, dumb terminals provide a simple, character-oriented conduit between the user and the host. Users can log on, run programs, read/write shared files, direct output to shared printers, and access shared databases. Furthermore, each terminal session functions independently from other terminal sessions because the arbitration between shared resources is performed deep inside the host operating system.

Terminal Services differ somewhat from the centralized host architecture. The primary difference is the graphical nature of the Windows 2000 Server operating system environment. Host environments have traditionally been character-oriented, requiring only a small amount of traffic (ASCII characters) to travel the communication lines between the host and the terminal or terminal emulator. With Terminal Services, all of the graphical screen output and related input/output (for example, from a mouse or keyboard) must flow between the desktop client (that is, the Windows-based Terminal or terminal emulator running on a computer) and the Windows 2000 Server running Terminal Services. This means for highly graphical and animated applications, a lot of information must travel over the network to the client device. Fortunately, the display protocol that operates between the Terminal Services client and the server optimizes this transmission and is completely transparent to the application developer. Additional information on the Microsoft Remote Desktop Protocol (RDP), the display protocol used by Terminal Services, can be found at the Microsoft Windows 2000 Web site.

Application Design Considerations

Another important difference between the host-based and Terminal Services environments is how applications that run in these environments must be designed. In a centralized host environment, applications must be developed specifically to run in that environment. With Terminal Services, applications designed for any Windows-based environment should work without having to be explicitly developed for the Terminal Services environment. Applications that run on Windows 2000 Server and Windows NT Server 4.0 should run without modification when Terminal Services is enabled.

This is important when you consider the implications of multiple users sharing a Windows 2000-based system simultaneously. Instead of different users running applications using their own computer hardware resources (such as CPU, memory, and disk) and local software resources (such as registry settings, preference files, and dynamic link libraries), Terminal Services users share hardware and software resources. For example, if two users run the same application in a Terminal Services environment, two copies of that application are started in the same system, each operating under a different user context. All of this is managed transparently by Terminal Services within the operating system.

Multiple users accessing the same set of applications in a common system can create contention:

- **Users contend for CPU time.** In the Terminal Services environment, each user has his or her own desktop and can run whatever applications are available to that desktop. However, all applications run by all users are contending for the central CPU resources available on the server machine. If one user runs a poorly written, CPU-intensive application, other users on that server may experience a visible loss of performance.
- **Users contend for disk access.** This is analogous to how users vie for disk access using traditional client/server network connections. In the Terminal Services environment, the input/output demands are more intense because users not only contend for access to applications and related application files, but also for server operating system disk access. For example, multiple users may be making different dynamic link library (DLL) calls at the same time or swapping between real and virtual memory areas. This single disk resource also represents a single drive structure. Using common areas instead of user-specific data directories can result in contention or collision.
- **Users contend for Random Access Memory (RAM).** Each user has an independent session, which he or she can fill with all the memory-intensive applications at his or her disposal. Some users may try to open as many applications on their desktop as they can, while others may take a more conservative approach and run only those applications that they need. Nonetheless, the needs of all the users are satisfied from the same core set of server memory resources.
- **Users contend for network access.** In any distributed processing environment, the network provides the pipeline for communication between the desktop and the servers. In the Terminal Services environment, the need for network access is more critical than in the traditional distributed client/server architecture because all desktop activity—graphical output and mouse/keyboard input—flows over network links between the desktop and the server. Without a functioning network connection to the server, a Terminal Services Client cannot operate at all. The network connection is used to communicate with each client, while still servicing the network needs of the applications and services.
- **Users contend for server-side hardware components.** Hardware components such as CD-ROMs, disk drives, serial ports, and parallel ports may be used on the server. When a user wants to access server-side hardware components, he or she may be competing with other users making similar requests. Sharing these traditionally non-shared components creates new considerations for both the users and for the applications that access them.

- **Users contend for access to global Windows 2000 objects and resources.** In the Terminal Services environment, users do not run individual copies of Windows 2000 Professional—some of the core operating system and applications components are cloned, but the remaining ones are shared among the users. Thus users are competing for access to the registry, the paging file, system services, and other global objects and resources.

Proper System Sizing

You can mitigate many of these points of contention by sizing the Terminal Services system with sufficient CPU, memory, and disk resources to handle the client demand. For example, a multiple processor configuration can maximize CPU availability. Installing extra physical memory can maximize memory accessibility. Finally, disk access performance can be made optimal by configuring multiple SCSI channels and distributing your operating system and application loads across different physical drives. Properly configuring a Terminal Services system is a critical element that improves application performance for the client. Deployment guidelines and capacity planning information for Terminal Services system administrators are available on the Microsoft Windows 2000 Web site.

Testing in a Typical Terminal Services Configuration

To ensure that an application installs and works properly in a Terminal Services environment, it is critical to test it in that environment. It is important to construct a typical usage scenario with the appropriate number of sessions running the application for a time period that simulates actual usage. In a typical personal computer desktop environment, the application and system may be shut down frequently enough to mask application problems such as memory leaks. Capacity planning guidelines for Terminal Services, which can be found the Windows NT Server Web site, demonstrate three user applications usage scenarios: light (task-oriented) user, medium (administrative) user, and heavy (knowledge) user. Developers should determine which scenario best fits the likely usage of their application, and follow the guidelines to configure a representative Terminal Services testing environment. Terminal Services client emulation tools with application scripting support are available in the Windows 2000 Server Resource Kit to assist in this testing.

Although hardware sizing and testing is an important part of creating a scalable Terminal Services environment, software considerations are equally important. In fact, fine-tuning an application can often considerably reduce resource competition and improve application performance for the user. The next section of this paper presents suggestions that you can easily implement to create programs that are optimized for the Terminal Services environment.

Conforming to Core practices

Managing and deploying applications from a central server provides benefits that you can enhance by fine-tuning applications that run when the Terminal Services feature is enabled in Windows 2000 Server. When you employ the programming practices suggested in this section, you ensure that an application will operate properly under Terminal Services. Keep in mind that following these practices does not inhibit the ability of your application to run in a non-Terminal Services environment. In fact, following these practices can often improve performance and increase compatibility with other services.

Core Practices

The following guidelines describe programmatic behavior that is necessary for an application to operate efficiently and effectively in the Terminal Services environment. There are three main areas where applications tend to run into trouble in a Terminal Services environment:

- Installation.
- Separation of user and global data.
- Programming for the multiuser.

In addition, Windows NT 4.0 Terminal Server Edition has some restrictions on DCOM usage.

Installation

Application installation is different when Terminal Services is enabled in Windows 2000 Server. The registry and .ini file mapping support that is built into Terminal Services allows applications that were not originally designed to run in a multiuser environment to run correctly under Terminal Services. This means that users should be able to execute these applications simultaneously and save whatever preferences the application allows for each of them. Of course, each user must have a unique home directory. If no home directory is specified for a user by the administrator, the user's home directory defaults to his or her user profile directory, \Wtsrv\Profiles\Username.

To enable each user to retain individual application settings, he or she must have a unique copy of the appropriate .ini files or registry entries. To accomplish this, Terminal Services replicates the .ini files and registry entries from a common system location to each user as necessary. For .ini files, this means that the .ini files in the system directory (`%systemroot%`) will be copied to each user's Windows directory. For registry entries, the registry entries will be copied from **HKEY_LOCAL_MACHINE \SOFTWARE \Microsoft \Windows NT\CurrentVersion\Terminal Server\Install\Software** to

HKEY_CURRENT_USER \Software.

In order for Terminal Services to replicate the necessary registry entries or .ini files for each user, the user must install the application in **Install** mode. This is accomplished by using **Add/Remove Programs** in **Control Panel**. Install mode may also be enabled from the command line after executing the **change user /install** command, though using **Add/Remove Programs** is preferable. If the administrator uses this function, application installation should properly allow for user-specific application settings. The built-in Windows Installer service included in Windows 2000 Server will be the best way for application developers to ensure correct installation. For more information on the Windows Installer service, see the current guidelines for the Windows 2000 Logo program

Disallow Installing Multiple Versions of Applications with Shared DLLs

Because many application versions share DLLs, only one version of an application can be run at a time. If multiple versions are installed on the system, it is very possible that different users will attempt to run various versions of the same application simultaneously. For example, both Microsoft Internet Explorer 3.*x* and Microsoft Internet Explorer 4.*x* share various DLLs that will fail to work properly when both versions are installed on the same server.

Application Installation or Compatibility Scripts

If an older application does not install properly, an Application Compatibility Script may be needed to correct installation problems with the registry or other issues. Scripts for many popular applications such as Microsoft Office are included with Windows 2000 Server Terminal Services. Additional information on these scripts can be found on the Microsoft Web site in the white paper "Using and Developing Applications Compatibility Scripts."

Separation of Local and Global Data

Use the HKEY_LOCAL_MACHINE Properly

In the Terminal Services environment, each user receives a **HKEY_CURRENT_USER** registry hive at logon time that stores user-specific information; however, all users share the **HKEY_LOCAL_MACHINE** hive. This means that any information placed in the **HKEY_LOCAL_MACHINE** hive affects all users, while information placed in the **HKEY_CURRENT_USER** hive affects only one user session.

Some applications make the assumption that one machine equates to one user, and they store user information in the **HKEY_LOCAL_MACHINE** hive. This practice can create serious problems in a multiuser environment. With this in mind, applications should properly separate global registry information from local (user) registry information, and store information in the correct hive.

Do Not Store Local Data Constructs in Global Locations

In addition to separating global and local information in the registry, global and local file-based data constructs should also be maintained separately. For example, user preference files should not be stored in the system directory (for example, Winnt) or program directory (for example, Program Files) structures. Instead, preference files or other user-specific local data constructs should be stored in the user's home directory or a user-specified directory.

This consideration also applies to temporary files used to store interim information (such as cached data) or to pass data on to another application. User-specific temporary files must also be stored on a per-user basis.

Programming for the Multiuser

Disallowing Multiple Instances of Some Applications

There are some types of applications that should only run with one instance on the server. Typically, these are applications that monitor or manage system resources, such as a disk administration program. These applications should check if they are already running and not initiate a second application process. In particular, if an application of this type polls a system resource continually, multiple instances of the application are not needed, and instead could seriously degrade system performance.

Resolve Memory Leaks

The danger of memory leaks is intensified in the Terminal Services environment. A memory leak in a program running in the traditional Windows client environment will eventually cause trouble, but may in fact be masked by the fact that the desktop device is turned on and off frequently and memory is thus cleared. In the Terminal Services environment, that same application can be run multiple times by multiple users, thus rapidly magnifying the effect of a memory leak.

Do Not Assume Computer Name or IP Address Equates to Single User

In the traditional distributed Windows-based client/server architecture, one user is logged on to one computer at a time; therefore, the computer name or Internet Protocol (IP) address assigned to either a desktop or server computer equate to one user. In the Terminal Services environment, the application can only see the IP or NetBIOS address of the Terminal Server.

Applications that use the computer name or IP address for licensing or as a means of identifying an iteration of the application on the network will not work properly in the Terminal Services environment because the server's computer name or IP address can really equate to many different desktops or users.

Do Not Assume the Windows Shell

Some applications assume that the Windows shell (including the browser) will be running and use these as resources in the application. If the administrator chooses, Terminal Services allows applications to run entirely without the shell or desktop. This feature is provided so administrators can lock down a client session and deny the user access to anything except a single application. This feature can be used in a task-based worker environment, where limiting end-user access to the desktop and file system reduces potential security or configuration problems, and reduces help-desk costs.

Do Not Assume Persistence of Files in Temp

Don't assume persistence of any files in the Temp folder beyond the current user session on any machine, because administrators can set a policy to delete everything in the Temp folder each time the user logs on. For example, if a recovery file that is regularly updated during an editing session is be stored in the Temp folder it may not be present to restore changes if an application crashes. Also, these types of files are clearly per-user, so they should be saved in the Application Data folder in the User Profile, where the appropriate file security is also in place so others cannot view the file. An administrator in an enterprise environment may also configure Terminal Services to save per user Application Data in a directory on a completely separate file server for recoverability reasons, particularly in a multi-server farm configuration.

Do Not Modify the GINA

Modifications to the Graphical Identification and Authentication (GINA) component are supported in Windows 2000 Terminal Services with the availability of Terminal Services APIs that allow for session management and client credential access. Windows NT Server 4.0, Terminal Server Edition does not support modifications to GINA. For more information on the Terminal Services APIs, see the white paper "Using and Understanding APIs for Terminal Server" in Microsoft TechNet.

Do Not Replace System Files

Because Terminal Server Edition is a modified version of Windows NT Server 4.0, it uses operating system files with the same file names as a Windows NT Server 4.0 without the Terminal Server enabled, but in fact the files may be very different. Replacing system files, such as the TCP/IP network stack, could result in serious system problems. The same situation can occur if an application replaces Windows 2000 Server operating system files while Terminal Services are enabled.

Negotiate Client/Server Connections Inside the System and Network

If your application is composed of a server component (such as a service) and client components (such as foreground applications) that communicate with the server component make sure that the server component can differentiate between multiple clients residing on the same system. To accomplish this, clients should establish communication with the server component through a well-defined global interface (for example, Remote Procedure Call or named pipes) and the server and client should negotiate a different communication channel for each user session.

This same client/server consideration applies to client applications running in the Terminal Services environment that need to establish connections to server components over the network. In this case, the client/server connection should use protocols that easily support this type of operation—such as TCP/IP where a different socket connection can be used for each client application. Applications should not assume that a single system connection equates to a single user session.

Support Customization Through User Profiles

Many applications can be customized at installation time to include or exclude specific features or components. This approach does not work in the Terminal Services environment, because users typically access a common set of program files and libraries. Therefore, if the administrator excludes a component during the initial setup of the application under the Terminal Services, all users will be prevented from accessing that component.

A better way of addressing customization—in both Terminal Services and in the traditional Windows client/server environment—is to enable feature selection through user profiles. To adopt this approach, determine feature selection or de-selection at run time based on settings in the current user's registry hives. In Windows 2000 Server, you can use the Group Policy MMC snap-in to configure which features are available for which users. Windows NT Server makes use of the Systems Policy Editor tool to specify settings for the user desktop.

In addition, the administrator in a Terminal Services environment must be able to override end-user selections by setting policies using the Group Policy MMC snap-in. or by setting System Policies in the Windows NT Server operating system. For example, CPU-intensive processes such as a background spelling checker, which might be fine to run in the background on a desktop PC, should be disabled automatically by the application in a Terminal Services environment so they do not degrade performance for other users. Application options should be designed to check Group Policies (or System Policies in Windows NT Server) first and default to those policies. This process should be transparent to the end user, who should not have to care if the application is running locally on the desktop or centrally from Terminal Services.

Multilingual and International Usage Scenarios

Windows 2000 Server with Terminal Services enabled offers multilanguage support. Using this capability, Terminal Services can simultaneously serve users in as many languages as are installed on the server. In Windows NT Server 4.0, Terminal Server Edition, a single Terminal Server does not have the ability to simultaneously host multiple system languages. (For example, in Windows NT, on a North American English version of Terminal Server, users can read and create documents using non-Western character sets (provided the required font files are installed), but the system uses English menus, dialog boxes, and other operating system functions.)

Consider the Peripheral Hardware Environment

The Terminal Services environment supports both traditional serial, parallel, and sound ports attached to the server, but does not natively support serial, parallel, and sound ports that are integrated into the client desktop system (except for keyboard and mouse). This means that the hardware environment can appear different to the application when it runs in different user contexts.

Automatic configuration of local client printing or drive resources from the server is not available in Windows NT Server 4.0, Terminal Server Edition. Configuration of local printers is available for clients running the RDP protocol in Windows 2000 Server. In Windows NT Server, Terminal Server Edition 4.0 print or file access to or from the client must use network redirection.

DCOM Support

Distributed Component Object Model (DCOM) is fully supported in Windows 2000 Terminal Services, so there are no special considerations there. In Windows NT Server 4.0, Terminal Server Edition, DCOM functionality is a subset of DCOM in the Standard Edition of Microsoft Windows NT Server 4.0. For this reason, some applications that are written for and function properly in a Windows NT Server 4.0 environment may not function properly when running on Windows NT Server 4.0, Terminal Server Edition. For more information regarding DCOM functionality in Terminal Server Edition, please see the white paper "Using DCOM with Windows NT Server 4.0, Terminal Server Edition and Terminal Services in Windows 2000" in Microsoft TechNet.

Optimizing 32-Bit Applications

An application can detect if it is being set up in or is running in the Terminal Services environment. Once that determination is made, the application can then optimize its behavior, based on the suggestions presented in this section. In some cases, the application may choose to alter its multiuser behavior when it detects it is running in a Terminal Services environment. In others, it may base changes on whether it is running on the console or a remote session.

Detecting If Terminal Services Are Enabled in Windows 2000

Since Terminal Services are a configurable service under Windows 2000, proper detection of the service is even more critical during both application installation and execution. In the Terminal Server 4.0 product, you could simply check a product suite key in the registry and look for a "**`Terminal Server`**" string. In Windows 2000 this is not the case because the "**`Terminal Server`**" string will always be included in the product suite key even when Terminal Services isn't enabled. Detection of Terminal Services in Windows 2000 must be done through a new set of product suite APIs (defined in WINBASE.H). See the **`IsTerminalServicesEnabled`** function in Appendix A for the proper code to use to determine if Terminal Services are enabled on the Windows 2000 Server.

Detecting Windows NT Server 4.0, Terminal Server Edition

Checking to see if your application is running in a Terminal Server 4.0 environment is possible using the product suite concept that was added to Windows NT 4.0 Service Pack 3. Appendix A contains an example function called **`ValidateProductSuite`** that can be used on Terminal Server 4.0 machines to determine if a particular product suite has been installed. Using the example function, a Terminal Server 4.0 system can be detected with the following code:

```
BOOL fIsTerminalServer40;
fIsTerminalServer40 = ValidateProductSuite("Terminal Server");
```

Detecting If Another Instance of an Application Is Running

For certain management tools or utilities, only one instance of the application should run on the server at a time. An example is a disk defragmenter, or a tool that monitors system resources such as hard drive space, memory, or I/O. Code can be used to detect if another instance of the application is already running under Terminal Services, so that initiation of another application process does not happen. See Appendix B for this code.

Detecting If an Application Is Running in a Console Session or Remote Session

A developer may want an application to behave differently when it is run on the console versus in a remote session. Animation, sounds, and peripheral access might be enabled for the console session. The code found in Appendix C can be used to detect what type of session the application execution request is being initiated from. Windows 2000 or Service Pack 4 for Windows NT Server 4.0, Terminal Server Edition is required for this code to function.

Sending a Message Box from a Service to a Remote Session

Many services attempt to display simple status information to the active user by displaying a message box. This is typically done by calling one of the Win32® MessageBox* APIs in conjunction with the MB_SERVICE_NOTIFICATION flag. However, on a Terminal Services-enabled system, these pop-up message boxes will usually be displayed on the console, because that is the session in which the service is running. This can be a problem for applications running in remote sessions that make Remote Procedure Call (RPC) calls into the service and then wait for the user to respond to the service's pop-up window. Because this pop-up window appears on the console, the application running in the remote session appears to hang.

In the Windows 2000 operating system, RPC calls into a service keep track of the session that initiated the RPC call. The Win32 MessageBox* APIs have been enhanced to look for this session information when the MB_SERVICE_NOTIFICATION flag is specified. Before making the MessageBox* API call, the service must impersonate the calling client. The code found in Appendix D can be used in RPC-based services that need to display pop up windows to their calling clients.

Tune Background Task Resource Consumption

Many applications use background tasks to provide a mechanism for handling low-priority tasks in a single-user environment. In the Terminal Services environment, the scheduler is generally optimized for interactive responsiveness of foreground tasks (more like Windows NT Workstation or Windows 2000 Professional than Windows NT or Windows 2000 Server), though under Windows 2000 Terminal Services the administrator can select this. Also, the demands of running many interactive sessions on a single server required some architectural changes to the kernel that resulted in a small system cache. This means that, when Terminal Services are enabled, one user's background task will compete for CPU cycles with another user's foreground tasks. When multiple users are running both foreground and background tasks, the CPU demands are much higher than when all users are running only foreground tasks. This situation typically arises when a Terminal Server is also being used as an applications server to host a client-server application such as Microsoft SQL Server™ or Microsoft Exchange, perhaps in a branch office environment. It is generally not a performance issue when the Terminal Server has only a few users attached, but could become an issue when the server is trying to serve many users. Microsoft recommends that Terminal Servers be dedicated to serving client applications, but customers sometimes wish to run the Terminal Server as a multi-purpose applications server. To maximize CPU availability for all users, application developers should create efficient background tasks that are not resource-intensive, or turn off background tasks when Terminal Services are running. In Windows 2000 Server, the administrator will be able to choose whether to give more priority to background or foreground processes even when Terminal Services are enabled. This allows more granular performance tuning by administrators for their particular network and applications configuration.

Tune Thread Usage

Threads provide a convenient way of allowing an application to maximize its usage of CPU resources in a system, especially in a multiple processor configuration. When this same technique is used in a Terminal Services environment, the thread demands are intensified. In the Terminal Services environment, multiple users are running multithreaded applications, and all of the threads for all of the users compete for the central CPU resources of that system. With this in mind, you should tune and balance application thread usage for the multiuser, multiprocessor Terminal Services environment. In Windows 2000 and future versions, operating system capabilities such as using I/O completion ports and thread pooling can help the system use multiple threads more efficiently. I/O completion ports allow multiple process threads that are reading a system port to be pooled or reduced to a single thread per processor instead of requiring a single thread per individual process.

Minimize Splash Screen Usage

Splash screens—graphical product, company, or user information that appears while an application is starting—work well in local video environments because of the speed of delivery. When that same splash screen is transmitted to a Terminal Services desktop client over the network, the transmission consumes extra network bandwidth and forces the user to wait before accessing the application. Limit the use of splash screens in order to speed up the application start-up process and to enhance the end-user experience. Particularly for remote access, reducing the size of the bitmap and testing at lower network speeds, for example, over a 28.8 K modem, ensures the best end-user experience.

Minimize the Use of Animation

When a program uses on-screen animation, that animation consumes both CPU time and video bandwidth. This has several ramifications in the Terminal Services environment. First, the resource consumption of animation affects other users; while staring at an animated icon on the screen, the individual user is depriving others of CPU access and network bandwidth. Second, the actual effect of the animation is compromised because the video output is being rerouted over the network. Therefore, to decrease network activity and to improve the end-user experience, keep the use of animation to a minimum. For optimal performance and feature-rich end user experience, do not run animated or bitmap-intensive applications in a Terminal Services remote session. Allow these applications to execute only when they detect a local Windows-based desktop operating system, such as Windows 2000 Professional or Windows NT Workstation 4.0, or when they are running on the console.

Minimize Direct Video Access

Many programmers are accustomed to the speed of a local video subsystem—a local subsystem is typically fast enough to prevent the user from seeing multiple images or windows being overlaid on the screen. This is not the case when the video stream flows over a network connection. In this case, the user will see (and be frustrated by) the amount of time it takes to render the final screen.

Applications should avoid direct input or output to the video display. If an application needs to read bits from the screen, it should maintain a separate, off-screen copy of the video buffer. Similarly, if an application needs to do elaborate screen output, such as overlaying several images to arrive at a final composite screen, the application should do that work in an off-screen buffer and then send the results to the actual video buffer.

Move User Input Routines to Foreground Applications

User input and prompts should be handled by foreground applications and not by services called by those applications. In the Terminal Services environment, if a user runs an application that calls a service that requests console input, the application will appear to stop or hang until the input is satisfied from the server console and not the desktop. Such problems are particularly arduous when Terminal Services is being used for Remote Administration tasks.

Enable Application Access for All Users

The automated setup procedure for many existing applications assumes that the application is being installed for a single user, and therefore updates the registry hive and desktop environment pertaining to just one user. If additional users need to access that application, either the entire package must be installed again or an administrator must manually copy information from the registry and desktop of one user to the other users.

In the Terminal Services environment, however, more than one user will access an application on a system. Also, the user accessing the application (the end user) is typically not the same person who installed the application (system administrator). Therefore it is appropriate to install applications into the default user environment common to all users when installing under the Terminal Services environment.

Enumerating System Resources

For applications that enumerate global system resources, such as a function that returns the number of running processes on the system, be aware that these applications will return the information for all sessions, not just the individual session. Also, this type of function should be limited so that multiple instances of it are not initiated (see discussion above), because they are typically resource-intensive and multiple instances will quickly degrade other users' performance.

Use Classes Where Possible

The Microsoft Foundation Class Library (MFC) has a long list of tried-and-true classes that perform a wide variety of tasks. Most of these classes work well in the Terminal Services environment, usually much better than do re-engineered solutions. A good example is the class available to provide context-sensitive Help text—Help text that appears on-screen when the mouse pointer moves over a button or menu item. If an application uses the MFC implementation to provide this feature, it works well on the Terminal Services client. But if the application implements this feature using dialog boxes or an alternate approach, the final result may not function well in the Terminal Services environment.

Legacy Applications

The Windows 2000 Server operating environment has specific requirements for hosting MS-DOS-based and 16-bit Windows-based applications. These requirements also apply to the Terminal Services environment—any MS-DOS or 16-bit Windows-based application that won't run on Windows 2000 Server or Windows NT Server 4.0 also won't run on Windows 2000 Server with Terminal Services enabled, or on Windows NT Server 4.0, Terminal Server Edition. When Terminal Services is enabled, the system is more sensitive to ill-behaved legacy applications than the traditional Windows client/server environment is, because a single ill-behaved application can affect all users on the Terminal Services network.

Predicting which legacy applications will work well in the Terminal Services environment and which will not is difficult. In most cases, the only way to determine the feasibility of a specific application is to test it in the Terminal Services environment. However, there are some application behaviors and origins that are known to be incompatible with or detrimental to the Terminal Services environment:

- **MS-DOS and 16-bit Windows-based applications require more RAM** than native 32-bit Windows-based applications when running on any Windows 2000 Server system. This is also true for the Standard Edition of Windows NT 4.0, and for Terminal Server Edition of that operating system. Windows runs an emulation layer on top of the 32-bit operating system. Although this memory requirement may not show up as performance degradation on a high-powered desktop computer running the latest Windows operating system with 64 MB of RAM, it may easily show up on a system running Terminal Services, because of the multiplier effect of many user sessions. For more information on system requirements for Terminal Services, see the Microsoft Windows 2000 Web site.
- **MS-DOS-based applications that cycle on device input.** MS-DOS-based applications that tightly loop for keyboard input, mouse activity, or other input device operations are often too CPU-bound to run effectively in the Terminal Services environment.
- **FoxPro® database for MS-DOS-based applications.** These applications tend to be CPU interrupt-intensive and drain processor resources away from other user sessions. As in the previous case, these applications often cannot run effectively in the Terminal Services environment.
- **MS-DOS-based print applications.** Many MS-DOS-based applications use print techniques that are not compatible with the Terminal Services environment. However, 16-bit Windows-based applications that use the standard Windows printing APIs function correctly.
- **MS-DOS and 16-bit Windows-based applications that internally mount NetWare drives.** Programmatic NetWare drive-mapping operations do not function under Terminal Services. On the other hand, programmatic uniform naming convention (UNC) mounts work correctly.
- **16-bit Windows-based applications that directly access .ini files**. Older Windows-based applications that directly access .ini files instead of using the standard APIs may not work when multiple users are concurrently running the application. To minimize compatibility issues and to maximize performance under Terminal Services Application, developers are strongly encouraged to port all MS-DOS or 16-bit Windows-based applications to the Win32 environment. Please refer to the MSDN Web site for additional information on porting 16-bit applications to the 32-bit Windows environment.

Conclusion

The amount of work it will take to optimize an application for a Terminal Services environment varies greatly from one application to another. If an application was designed to operate in a multiuser environment, only minor changes will be needed to turn it into an application that is optimized for Terminal Services in Windows 2000 or Windows NT Server, Terminal Server Edition, version 4.0. Single-user, desktop applications may require more modification.

This document has presented a variety of guidelines and suggestions for fine-tuning applications for the Terminal Services environment. All of these guidelines and suggestions should be taken into consideration when you develop 32-bit applications; however, you should remember these three points in particular about the Terminal Services environment:

- Applications must distinguish between global and local (user-specific) data structures, including registry keys. All data structures should be locked during access and stored in an appropriate location.
- The Terminal Services environment uses more network resources than a client/server or distributed computing environment. In the Terminal Services environment, all graphical output and keyboard/mouse input flows over the network. Overlaid images, on-screen animation, and splash screens take longer to display on the client computer and can slow down the user experience.
- In the Terminal Services environment, all application processing occurs on the server. Applications that monopolize CPU, RAM, and hard drive resources or otherwise assume a single-user environment will not perform well in a Terminal Services environment when multiple sessions are running.

By taking these points into consideration, developers can make the most of their applications in both the traditional Windows-based client/server environment and in the Windows 2000 Terminal Services environment.

For More Information

To read any of the white papers referenced in this paper, or to find the latest information on Windows 2000 Terminal Services and Windows NT Server 4.0, Terminal Server Edition, check out Microsoft TechNet or the Microsoft World Wide Web site.

Appendix A

Detecting If Terminal Services Is Enabled

The following is the code for IsTerminalServicesEnabled that can be used to detect whether Terminal Services is enabled. It is compatible with all Win32 platforms. (Note, if your application is designed to run only on the Windows 2000 platform, a simplified version of this code is provided below.)

```
// This function compares the passed in "suite name" string
// to the product suite information stored in the registry.
// This only works on the Terminal Server 4.0 platform.
```

```
BOOL ValidateProductSuite (LPSTR SuiteName)
{
BOOL rVal = FALSE;
LONG Rslt;
HKEY hKey = NULL;
DWORD Type = 0;
DWORD Size = 0;
LPSTR ProductSuite = NULL;
LPSTR p;

Rslt = RegOpenKeyA(
HKEY_LOCAL_MACHINE,
"System\\CurrentControlSet\\Control\\ProductOptions",
&hKey
);

if (Rslt != ERROR_SUCCESS)
goto exit;

Rslt = RegQueryValueExA( hKey, "ProductSuite", NULL, &Type, NULL, &Size );
if (Rslt != ERROR_SUCCESS || !Size)
goto exit;

ProductSuite = (LPSTR) LocalAlloc( LPTR, Size );
if (!ProductSuite)
goto exit;

Rslt = RegQueryValueExA( hKey, "ProductSuite", NULL, &Type,
(LPBYTE) ProductSuite, &Size );
if (Rslt != ERROR_SUCCESS || Type != REG_MULTI_SZ)
goto exit;

p = ProductSuite;
while (*p)
{
if (lstrcmpA( p, SuiteName ) == 0)
{
rVal = TRUE;
break;
}
p += (lstrlenA( p ) + 1);
}

exit:
if (ProductSuite)
LocalFree( ProductSuite );

if (hKey)
RegCloseKey( hKey );

return rVal;
}
// This function performs the basic check to see if
// the platform on which it is running is Terminal
// services enabled. Note, this code is compatible on
// all Win32 platforms. For the Windows 2000 platform
// we perform a "lazy" bind to the new product suite
// APIs that were first introduced on that platform.

BOOL IsTerminalServicesEnabled( VOID )
{
BOOL bResult = FALSE; // assume Terminal Services is not enabled

DWORD dwVersion;
OSVERSIONINFOEXA osVersionInfo;
DWORDLONG dwlConditionMask = 0;
HMODULE hmodK32 = NULL;
HMODULE hmodNtDll = NULL;
typedef ULONGLONG (*PFnVerSetConditionMask)(ULONGLONG,ULONG,UCHAR);
typedef BOOL (*PFnVerifyVersionInfoA) (POSVERSIONINFOEXA, DWORD, DWORDLONG);
PFnVerSetConditionMask pfnVerSetConditionMask;
PFnVerifyVersionInfoA pfnVerifyVersionInfoA;

dwVersion = GetVersion();

// are we running NT ?
if (!(dwVersion & 0x80000000))
{
// Is it Windows 2000 (NT 5.0) or greater ?
if (LOBYTE(LOWORD(dwVersion)) > 4)
{
// In Windows 2000 we need to use the Product Suite APIs
// Don't static link because it won't load on non-Win2000 systems

hmodNtDll = GetModuleHandleA( "ntdll.dll" );
if (hmodNtDll != NULL)
{
pfnVerSetConditionMask =
(PFnVerSetConditionMask )GetProcAddress( hmodNtDll, "VerSetConditionMask");
if (pfnVerSetConditionMask != NULL)
{
dwlConditionMask =
(*pfnVerSetConditionMask)(dwlConditionMask, VER_SUITENAME, VER_AND);
hmodK32 = GetModuleHandleA( "KERNEL32.DLL" );
if (hmodK32 != NULL)
{
pfnVerifyVersionInfoA =
(PFnVerifyVersionInfoA)GetProcAddress( hmodK32, "VerifyVersionInfoA") ;
if (pfnVerifyVersionInfoA != NULL)
{
ZeroMemory(&osVersionInfo, sizeof(osVersionInfo));
osVersionInfo.dwOSVersionInfoSize = sizeof(osVersionInfo);
osVersionInfo.wSuiteMask = VER_SUITE_TERMINAL;
bResult = (*pfnVerifyVersionInfoA)(
&osVersionInfo,
VER_SUITENAME,
dwlConditionMask);
}
}
}
}
}
else
{
// This is NT 4.0 or older
bResult = ValidateProductSuite( "Terminal Server" );
}
}

return bResult;
}
```

Here is a sample program that calls the IsTerminalServicesEnabled function in order to display a pop up window indicating if Terminal Services is enabled.

```
int WINAPI WinMain(
HINSTANCE hInstance, // handle to current instance
HINSTANCE hPrevInstance, // handle to previous instance
LPSTR lpCmdLine, // pointer to command line
int nCmdShow // show state of window);
)
{
BOOL fIsTerminalServer;
UNREFERENCED_PARAMETER (hInstance);
UNREFERENCED_PARAMETER (hPrevInstance);
UNREFERENCED_PARAMETER (lpCmdLine);
UNREFERENCED_PARAMETER (nCmdShow);
fIsTerminalServer = IsTerminalServicesEnabled();
if (fIsTerminalServer)
MessageBoxA( NULL, "Terminal Services is running.", "Status", MB_OK );
else
MessageBoxA( NULL, "Not a Terminal Services box.", "Status", MB_OK );
return 0;
}
```

Simplified Detection for Windows 2000 Only Applications

If your application or service runs exclusively on the Windows 2000 operating system, you can simplify the IsTerminalServicesEnabled function by directly linking with the Windows 2000 product suite API VerifyVersionInfo (defined in WINBASE.H) and specifying a wSuiteMask of VER_SUITE_TERMINAL (defined in WINNT.H). This simplified code is as follows:

```
#include <windows.h>
#include <stdio.h>
```

```
// This code will only work on the Windows 2000 platform

BOOL IsTerminalServicesEnabled(void)
{
OSVERSIONINFOEX osVersionInfo;
DWORDLONG dwlConditionMask = 0;

ZeroMemory(&osVersionInfo, sizeof(OSVERSIONINFOEX));
osVersionInfo.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
osVersionInfo.wSuiteMask = VER_SUITE_TERMINAL;

VER_SET_CONDITION( dwlConditionMask, VER_SUITENAME, VER_AND );

return VerifyVersionInfo(
&osVersionInfo,
VER_SUITENAME,
dwlConditionMask
);
}
```

Detecting Terminal Services Remote Administration Mode

In the Windows 2000 operating system, Terminal Services can be enabled in two modes: Application Server and Remote Administration. Remote Administration mode is a two client limited form of Terminal Services that is specifically intended for remote administrative access to a server. Terminal Services enhanced application compatibility code is disabled in this mode in order to simplify installation of server class applications. Note that all of the previous code example will return TRUE because they only check to see that Terminal Services is enabled. The following code distinguishes the Remote Administration mode of Terminal Services. Note that this is the same Windows 2000 example, except that the wSuiteMask value is set to VER_SUITE_SINGLEUSERTS (defined in WINNT.H).

```
BOOL IsTerminalServicesRemoteAdminEnabled( VOID )
{
OSVERSIONINFOEX osVersionInfo;
DWORDLONG dwlConditionMask = 0;

ZeroMemory(&osVersionInfo, sizeof(OSVERSIONINFOEX));
osVersionInfo.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
osVersionInfo.wSuiteMask = VER_SUITE_SINGLEUSERTS;

VER_SET_CONDITION( dwlConditionMask, VER_SUITENAME, VER_AND );

return VerifyVersionInfo(
&osVersionInfo,
VER_SUITENAME,
dwlConditionMask
);
}
```

Appendix B

Disallowing Multiple Instances of Some Types of Applications

The following code example shows how to limit your application to running only a single instance under Terminal Services. The code is Win32-compatible, so it can run on all platforms. The key is to use a mutex (mutual exclusion) object to determine if the application is already running. The old practice of using a window handle does not work under Terminal Services, because window handles are specific to a session, thus there are no system global window handles. The code uses the Terminal Services multisession object manager Global\ prefix to force the mutex into the system-wide *global* name space.

```
//
// Global Variables
//
BOOL g_fIsTerminalServer = FALSE;
TCHAR g_szAppName[] = TEXT("Generic");
HANDLE g_hAppRunningMutex = NULL;

//
// IsAppAlreadyRunning
//
// This routine check to see if the application is already running.
// The fOnePerSystem flag is used for Terminal Server, thus allowing
// you to limit the running instances to one per system or one per
// user session.
//
// NOTE: The g_hAppRunningMutex handle must remain open while your
// application is running.
//
BOOL IsAppAlreadyRunning( PCTSTR pszAppName, BOOL fOnePerSystem )
{
TCHAR szMutexName[MAX_PATH];

ASSERT(pszAppName != NULL);
ASSERT(g_hAppRunningMutex == NULL);

// Create a mutex in the global name space to see if an instance
// of this application is already running. If so, exit the app.
*szMutexName = TEXT('\0');

if (fOnePerSystem && g_fIsTerminalServer) {
//
// We're running on Terminal Server, so prefix the mutex name
// with Global\ to force it into the system global name space
//
lstrcpy(szMutexName, TEXT("Global\\"));
}
lstrcat(szMutexName, g_szAppName);
lstrcat(szMutexName, TEXT(" is running"));
g_hAppRunningMutex = CreateMutex(NULL, FALSE, szMutexName);
if (g_hAppRunningMutex != NULL) {
//
// Make sure we are the only process with a handle to our named mutex.
//
if (GetLastError() == ERROR_ALREADY_EXISTS) {
// The app is already running
CloseHandle(g_hAppRunningMutex);
g_hAppRunningMutex = NULL;
}

return (g_hAppRunningMutex == NULL);
}


//
// Cleanup routine. This should be called right before the application
// exists. Once this routine closes the g_hAppRunningMutex handle, then
// another instance of your application will be allowed to run.
//
VOID LetAnotherInstanceRun( VOID )
{
if (g_hAppRunningMutex != NULL) {
CloseHandle(g_hAppRunningMutex);
g_hAppRunningMutex = NULL;
}
}


//
// Example of how these routined would be called from WinMain
//
WinMain( ... ) {

//
// First, determine if we're running on a TS enabled system.
//
g_fIsTerminalServer = IsTerminalServicesEnabled();

//
// Check to see if another instance is running
//
if (IsAppAlreadyRunning(g_szAppName, TRUE)) {
// Display message box to user to let them know
// that only one instance is allowed.
return;
}

....

//
// Close the App's mutex, thus allowing another instance
// to run.
//
LetAnotherInstanceRun();
}
```

Appendix C

Detecting If an Application Is Running in a Console Session or Remote Session.

This code can be used to detect what type of Terminal Services session the application execution request is being initiated from. The Windows 2000 operating system or Service Pack 4 for Windows NT Server 4.0, Terminal Server Edition is required. On all other Win32 platforms, this code will always indicate that the process is running on the console. The following code is compatible with all Win32 platforms, however, the SM_REMOTESESSION value (defined in WINUSER.H) is only defined when you compile with a WINVER value >= 5.0.

```
if (GetSystemMetrics(SM_REMOTESESSION))
{
// App is running on a remote session.
} else {
// App is running on the console.
}
```

Appendix D

Displaying a Pop-up Message Box in a Remote Session from a Service

In the Windows 2000 operating system, the MessageBox* APIs check for the presence of an impersonation token when the MB_SERVICE_NOTIFICATION flag is specified. The impersonation token contains the session id of the client that is being impersonated and that is the session in which the pop-up message box is displayed. RPC based services can use this new MessageBox* feature in conjunction with client impersonation in order to make sure the message box is displayed in the client's session. The following code demonstrates how an RPC service can take advantage of this feature. (Note, this works with any form of impersonation, not just with RPC services.)

```
RPC_STATUS RpcStatus;
int iResult;

//
// Impersonate the client. We do this before we call MessageBox*
// because the impersonation token contains the session id of
// our calling client.
//
RpcStatus = RpcImpersonateClient( NULL );
if( RpcStatus != RPC_S_OK ) {
return( STATUS_CANNOT_IMPERSONATE );
}

//
// Now that we're impersonating, call the MessageBox* API with
// the MB_SERVICE_NOTIFICATION flag. This will redirect the popup
// to the client's session.
//
iResult = MessageBox(
HWND,
TEXT("Message to display to user in remote session"),
TEXT("Caption of Message Box"),
MB_OK | MB_SERVICE_NOTIFICATION
);

//
// Stop Impersonating
//
RpcRevertToSelf();
```

09/99
