2

The New Request Architecture

In the earliest versions of IIS, web application code ran in-process within the web server itself, and unloading or replacing an application often meant restarting the entire web service. It also meant that if an application had problems, it typically took IIS down with it. Even worse was that errors in a web application could end up corrupting other web applications. Starting with IIS 4, support for out-of-process applications was introduced. However, this meant performance degradation as requests were first sent to the main IIS process, then to the out-of-process application, and finally back to the main IIS process again. With the release of IIS 6.0, the entire underlying request processing architecture has been redesigned and vastly improved over previous versions, improving both reliability as well as performance.

In IIS 6.0, the components that are critical to the proper functioning of web services are isolated from all web applications. All requests are handled by the kernel-mode HTTP handler **`HTTP.sys`**. All applications that process the requests are handled by the **Web Administration Service** (**WAS**). All web applications now run out-of-process, but without the performance penalty, as the requests are routed to the appropriate process directly from `HTTP.sys`. The `HTTP.sys` and the WAS, reside in their own separate process spaces and do not allow third-party code to be loaded into them, preventing a misbehaving web application from affecting the web services.

IIS 6.0 has the capability to isolate third-party code into separate **application pools** of one or more **worker processes**, to avoid impacting the entire server when an application malfunctions.

A **worker process** is simply a user mode application, which processes HTTP requests such as requests for a static page, invoking an ISAPI filter or extension, running a CGI, or executing application code. Worker processes are implemented as executables called `w3wp.exe` and are controlled by WAS. The `HTTP.sys` listener routes requests and responses to and from worker processes.

An **application pool** is a group of web applications that share one or more worker processes. Application pools allow configuration information to be applied separately to one or more web applications and the worker processes that serve them. Each application within an application pool shares one or more worker processes.

With this new architecture, IIS 6.0 automatically detects application crashes, memory leaks, and other errors. When these conditions occur, IIS 6.0 provides fault tolerance as well as the ability to restart the worker processes as necessary. IIS 6.0 also takes the preventive step of recycling worker processes, thereby avoiding memory leaks and performance degradations before they build up. In these cases, IIS 6.0 continues to queue requests without interrupting the user experience.

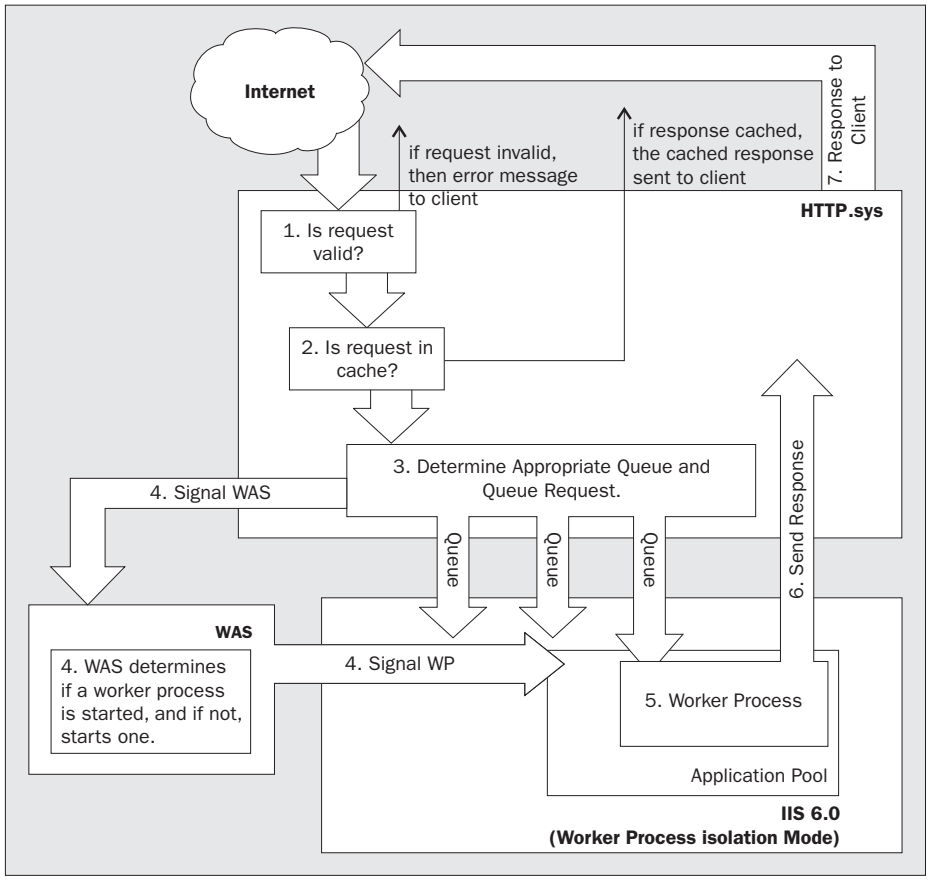
In this chapter, we will examine the features and components of the new request architecture, and discuss the following topics:

- Request flow architecture
- Application Isolation Modes
- Kernel-mode HTTP listener (`HTTP.sys`)
- Web Administration Service (WAS)
- Application Pools
- Application Health Monitoring
- Web Gardens

Request Flow Architecture

We will now take a look at how a request is processed and what path it follows within IIS. The path of the request has been illustrated in the following diagram:

Figure 1



The request flow occurs as follows:

1. When a request arrives, `HTTP.sys` determines if the request is initially valid. If it is not, an Invalid Request error code is sent back to the client and the connection is closed.

2. If the request is valid, `HTTP.sys` then checks to see if the response can be found in its kernel mode cache and, if it is present in the cache, `HTTP.sys` sends the response.

3. If the response is not in the cache, `HTTP.sys` then determines the appropriate application pool to receive the request, and puts the request into that application pool's request queue. All processing up to this point is handled in the kernel mode.
4. If there is no worker process running for the queue, `HTTP.sys` notifies WAS to start a worker process. WAS is the first user mode application becoming involved in the process.
5. Worker processes check their queues, and pick up the requests that are waiting for processing.
6. After processing, the worker process sends the response back to `HTTP.sys`
7. `HTTP.sys` sends the response back to the client and, if configured to do so, logs the request.

We will describe the key components, `HTTP.sys` and WAS, and their roles in the request path flow, later in the chapter. However, before that we need to understand the isolation modes offered by IIS 6.0:

- **Worker Process Isolation Mode**
- **IIS 5 Isolation Mode**

All applications are executed in either one of the two modes.

Application Isolation Modes

While both modes still use `HTTP.sys` as their listener, they are distinctly different in their operation. The isolation mode you select will have an impact on both performance and reliability, and will determine which features are available to you. Worker Process Isolation Mode is the recommended mode of operation for IIS 6.0, as it offers increased reliability through better isolation of applications. You should use worker process isolation mode unless there is a genuine issue of compatibility that will force you to use IIS 5 isolation mode; such as the need for **ISAPI raw read filters**, a feature that is not supported in worker process isolation mode.

IIS 6.0 Worker Processes Isolation Mode

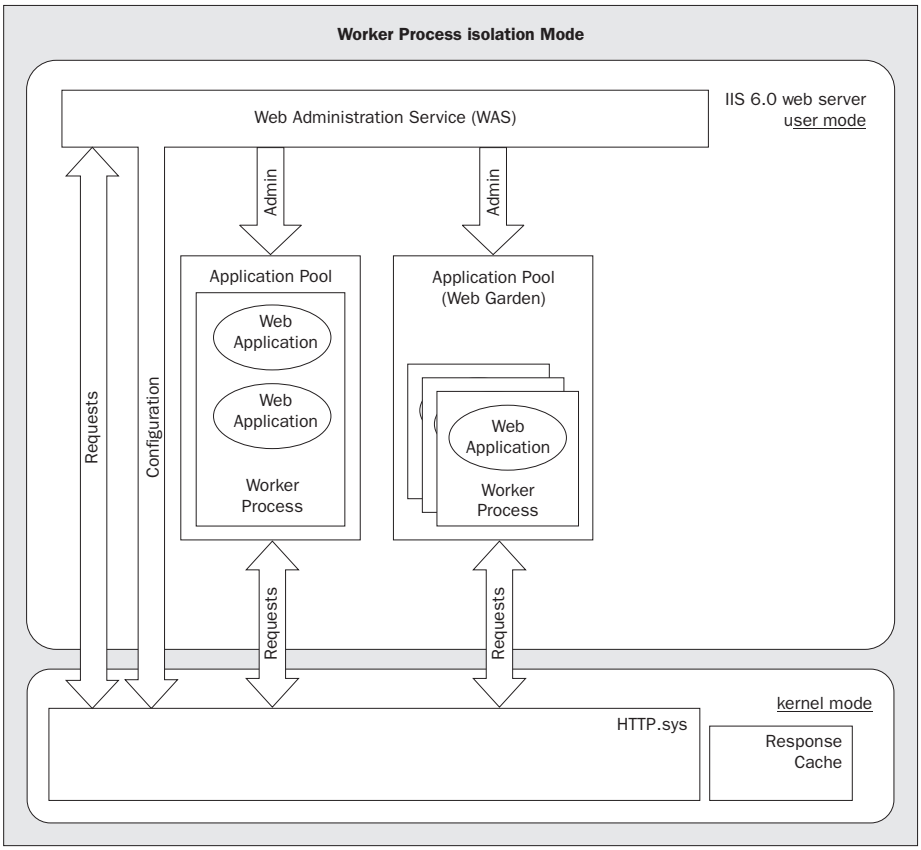
The new IIS 6.0 mode is called **Worker Process Isolation Mode**, because all applications are isolated as worker processes and hosted by a **Web Application Manager** (**WAM**) object, instead of `dllhost.exe`, as in previous versions. In IIS 5.0, you were limited to a single application pool. In IIS 6.0, you may have many application pools of worker processes, each worker process being an instance of an application called `w3wp.exe`. You might think of each worker process as its own **World Wide Web Publishing Service** (**W3SVC**), able to load and host applications in-process. In essence, anything that was done in W3SVC (in IIS 5) is now done by the worker processes in IIS 6.0.

How Worker Process Isolation Mode works

IIS 6.0 creates a separate worker process for each application pool, configuring each application to work in a separate isolated process, and thus ensuring that applications do not interfere with each other. An ISAPI application that would have crashed in IIS 5, would have taken down the whole server. In IIS 6.0 worker process isolation mode, such a crash would be detected and handled, with the rest of the server completely unharmed by any problem. The WAS doesn't restart worker processes until there's a request for the web application, thereby preserving resources until a worker process is actually needed.

The following diagram illustrates the working of worker process isolation mode:

Figure 2



`HTTP.sys` resides in kernel mode. In user mode, WAS manages application pools and the configuration of both `HTTP.sys` as well as application pools.

The Benefits of Worker Process Isolation Mode

IIS 6.0 Worker Process Isolation Mode improves upon the previous process models by providing increased scalability, reliability, and manageability. In this mode, there can be multiple worker processes available to handle requests, and each worker process is multi-threaded and capable of handling multiple user requests.

With multiple worker processes, **Web Gardens** can be created, further increasing scalability on multi-processor machines. Like a **Web Farm**, which is comprised of a number of similar machines working to balance request loads, a Web Garden is a single machine, balancing request loads across multiple processors. The key difference in IIS 6.0 is that worker processes can be assigned to individual processors. This way, if a worker process is blocked for a period of time, on a request that takes time (such as a database query), other worker processes are available on other processors. Worker process recycling may be configured to occur on a round-robin basis between processors, eliminating the short downtime during a recycle.

With all user code removed from the HTTP listener and WAS, IIS 6.0 provides increased reliability through isolation of potentially harmful code. Additionally, application health monitoring ensures that malfunctioning or misbehaving code is recycled or shut down gracefully as appropriate. Grouping similarly configured applications into application pools allows for better control over configuration, increasing manageability, and allows for control down to the namespace/application level.

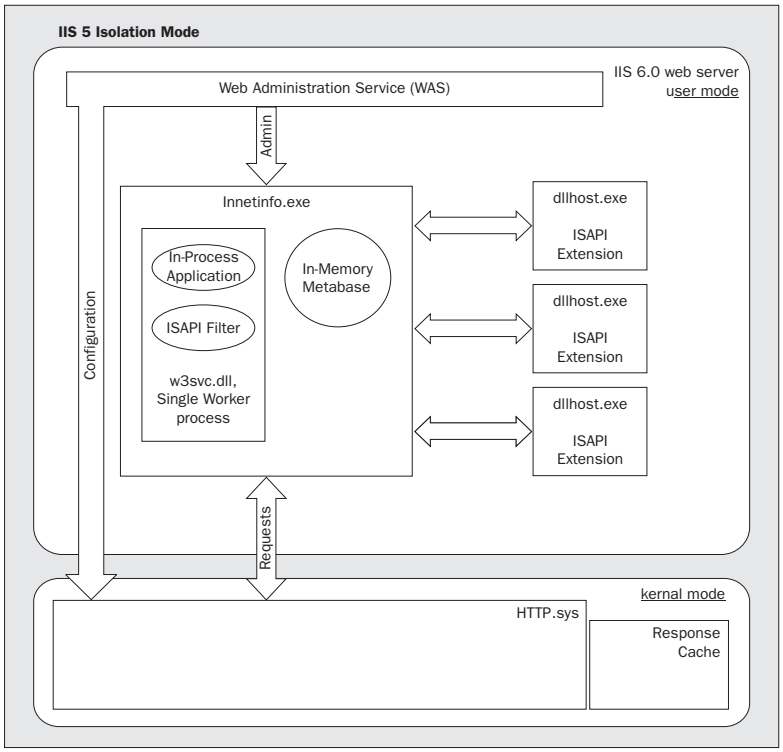
IIS 5.0 Isolation Mode

There are a number of reasons that the new worker process isolation mode may not be right for you. Your application may not be able to operate when there are multiple instances running, or may not be able to deal with the possible conditions in which the session state may be lost. While both of these issues can be addressed with configuration changes in worker process isolation mode, an insurmountable problem is that in many cases there are existing applications that make use of ISAPI's read raw data feature, which does not work in IIS 6.0 worker process isolation mode.

If you do not intend to upgrade to the new way of doing things, you will want to use **IIS 5.0 Isolation Mode**, which provides backwards compatibility for applications that require an environment similar to IIS 5. As in IIS 5, applications run as part of IIS process (in-process, inside of `inetinfo.exe`) or in separate process (`dllhost.exe`), but there is no isolation between web applications. `HTTP.sys` processes requests for this mode in the same way as IIS 6.0 worker process isolation mode.

The following diagram illustrates the working of IIS 5 isolation mode:

Figure 3



In the previous diagram, note the absence of all but one worker process. In this mode, `HTTP.sys` creates only one request queue. This was the drawback of IIS 5 isolation mode in that it provided only one application pool.

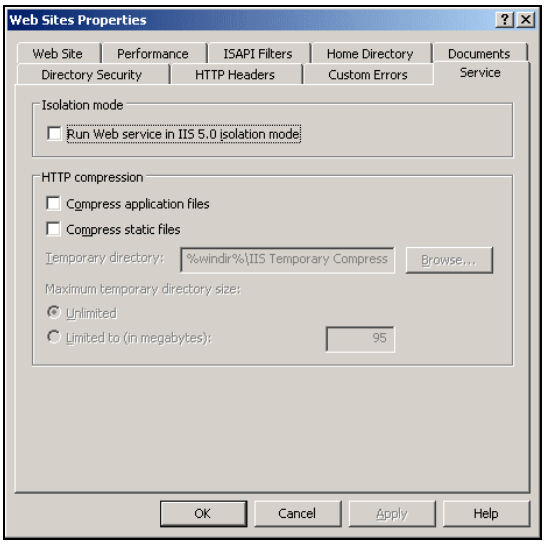
Which Mode?

IIS 6.0 will initialize itself in either worker process isolation mode or IIS 5 isolation mode, but not both. In other words, it is not possible to run some web applications in worker process isolation mode and others in IIS 5 isolation mode. New installations of IIS 6.0 will start in IIS 6.0 worker process isolation mode, while upgrades of any older versions of IIS (including version 5 and below) will start in IIS 5.0 isolation mode by default. Upgrades from IIS 6.0 will retain the mode previously used. If in the process of an upgrade, the mode is switched to worker process isolation mode, all applications will be placed in the single default application pool. Note that when the applications are so moved, their isolation configuration will be different from their previous configuration, and may need to be manually changed. See the section *Application Isolation and Performance* later in the chapter.

Switching Between Modes

You can change the application isolation mode, either with the **Microsoft Management Console** (**MMC**) snap-in or the scripting interfaces, and you can change it back and forth at any time. For example, if you are running an application that uses read raw filters, you can set the mode to IIS 5 isolation mode while you redesign your application. Once the redesign is done, you can set it back to worker process isolation mode. The application isolation mode is controlled through the `IIS5IsolationModeEnabled` property in the metabase (the IIS 6.0 configuration data structure). Setting this property to false will enable worker process isolation mode, and setting it to true will enable IIS 5 isolation mode. Changing this value requires a restart of the W3SVC service before the change is enacted.

You can change the isolation mode by starting the **IIS Manager** and accessing the Properties for web sites. The mode can be set in the property page under the Service tab:



Architectural Considerations of Worker Process Isolation Mode

There are a number of considerations to keep in mind when designing applications (or porting and configuring existing applications) for worker process isolation mode. State management becomes more complicated in an environment where processes may be recycled, as one process may be shut down in favor of a new one. Processes may be running multiple times, which adds new challenges for applications that must be prepared to operate in a multi-instanced environment. Worker process isolation mode also presents changes to the way that ISAPI filters operate, including the absence of read raw filters. We'll also take a look at special considerations for ASP.NET, and the implications of application isolation on performance.

State Management

When a worker process times out due to idle processing and is automatically shut down, any session state information stored in that process may be lost. Recycling, which causes the worker process to be restarted, may result in lost state as well. Applications should persist any state externally (such as in a database, or the ASP.NET session service). If your session state management code cannot be modified, IIS should be configured to run in a mode that does not threaten to lose state, including disabling recycling and idle timeout of worker processes.

Multi-Instancing

Multi-instancing, or two or more instances of a process running simultaneously, can pose problems for applications not prepared for this occurrence. Applications that use kernel objects such as mutexes, must be prepared for other instances to be accessing the same object, both within the same process as well as other processes. Applications that implement custom logging modules must be prepared for other instances to be accessing the same log. To avoid multi-instancing issues, you will not only need to ensure that there is only one worker process per application pool, but you will also need to disable overlapped recycling, which could result in the existence of two worker processes during the overlapped portion of an application pool recycle.

ISAPI Filters

In IIS 5.0, ISAPI filters ran in `inetinfo.exe` as LocalSystem and were guaranteed to be single-instanced. In worker process isolation mode, this is no longer the case. ISAPI filters may be multi-instanced, have different process identities, and are subject to recycling. Perhaps most importantly, the worker process isolation mode does not support `SF_READ_RAW_DATA` and `SF_SEND_RAW_DATA`. If you have a filter that registers for these notifications, and cannot be modified, you will have to run in IIS 5 isolation mode.

To resolve these issues in IIS 6.0, it is recommended that you use ISAPI extensions instead of filters. With the addition of wildcard scriptmaps and the `HSE_REQ_EXEC_URL` server support function, ISAPI extensions may now be used in the same role as traditional ISAPI filters. Additionally, ISAPI extensions are asynchronous (as opposed to the synchronous nature of ISAPI filters), which will provide significant performance gains. For more information on filters and extensions, see *Chapter7*.

Special considerations for ASP.NET

ASP.NET was originally released for use with IIS 5.0 and used its own process model. When ASP.NET is run in IIS 5 isolation mode, it will use its own process model and configuration settings as provided in the `machine.config` file. When ASP.NET runs on IIS 6.0, however, it uses the worker process isolation mode, disabling its own process model. This means that if your ASP.NET application has specific configuration settings in the `<processModel>` section of its `machine.config` file, most of those settings will be ignored in favor of the worker process isolation mode settings. The exceptions to this rule are the `maxIOThreads` and `maxWorkerThreads` entries, which will be read and used by IIS. The `maxIOThreads` value will control the number of threads the worker process will use to receive asynchronous requests from `HTTP.sys`, and the `maxWorkerThreads` value will set the number of application threads used by the ASP.NET ISAPI.

Application Isolation and Performance

Not to be confused with the concept of application isolation modes, **application isolation** is the separation of applications by process boundaries, which prevents them from interfering with one another. As we have seen, application isolation in worker process isolation mode is accomplished via application pools. In IIS 5 isolation mode, you can configure isolation using the `AppIsolated` property setting for the application, selecting in-process, pooled, or high isolation. This is very similar to the options available to you in IIS 5.0. In either pooled or high isolation mode, there is a performance hit caused by remote procedure calls necessary between `inetinfo` and the WAM object (an example of which would be retrieving an ISAPI Server Variable). This performance hit is not present in worker process isolation mode, as applications are loaded in-process to `w3wp.exe`.

Here's a comparison between the two modes with respect to the features provided:

Feature	Worker Process Isolation Mode	IIS 5 Isolation Mode
Basic Request/Response	Yes	Yes
Runs ISAPI Filters and ISAPI Extensions	Yes, as Worker Processes	Yes, In-process (`inetinfo.exe`) as well as out-of-process (`dllhost.exe`)
Worker Process Management	Yes, in WAS. Processes run as `w3wp.exe`	No
Application Pooling	Yes (multiple pools)	Limited (one pool only)
Application Recycling	Yes	No

Feature	Worker Process Isolation Mode	IIS 5 Isolation Mode
Web Gardens	Yes	No
Health Monitoring	Yes	No
Debugging	Yes	Limited
Processor Affinity	Yes	No
Performance Tuning	Yes	Limited
`HTTP.sys` Configuration	Yes, in WAS	Yes, in WAS
Out-of-Process ISAPI	`w3wp.exe`	`dllhost.exe`
ISAPI Filters	`w3wp.exe`	`inetinfo.exe`
FTP, NNTP, and SMTP	`inetinfo.exe`	`inetinfo.exe`

Now that we have taken a comprehensive look at the isolation modes offered by IIS 6.0, we will describe the two components of the request architecture: `HTTP.sys` and Web Administration Service.

HTTP.sys

The **Hyper Text Transfer Protocol** (**HTTP**) stack (`HTTP.sys`) is a new kernel mode driver, listening directly at the TCP/IP level, and is the sole channel for HTTP requests to IIS. In IIS 5, HTTP requests were served by the **Winsock/afd.sys** components, which had difficulty sharing ownership of port 80, the port for HTTP and port 443, the port for **Secure Socket Layers** (**SSL**). Furthermore, since there were several teams at Microsoft who all implemented their own version of the HTTP server-side stack, there were a number of places where bugs could be introduced into the HTTP process.

For Windows Server 2003 (which includes the IIS 6.0), the networking team decided to unify those efforts and develop a server-side HTTP listener that would offer a pure "request and response" kernel mode API. **Kernel mode** describes the privileged processor mode in which the NT-based operating system executive code runs. The code executing in the kernel mode has access to critical operating system resources, such as system memory and hardware. The boundary between kernel mode and **user mode** (in which applications run) is designed to protect the operating system from bugs introduced by user code. In other words, a misbehaving application in user mode will not interfere with the operating system or its ability to support other applications.

HTTP.sys is a new part of the networking subsystem in Windows Server 2003, and is available not only to IIS 6.0, but also other components as well. Having a separate HTTP service allows other applications that utilize HTTP to benefit from a dedicated high-performance HTTP stack. HTTP.sys is responsible for all TCP connection management, request routing, text-based logging, caching, and **Quality-of-Service** (**QoS**) functions, including bandwidth management, connection limits and timeouts, and queue length limits. Since HTTP.sys is not processing requests other than routing them to the correct consumer, no application-specific code is ever loaded into the kernel mode. This means that developers don't need to worry about errors being introduced into the kernel mode that cause an appearance of the dreaded blue screen.

Kernel-level Queuing

HTTP requests come into HTTP.sys (which is responsible for all connection management) and are routed to the appropriate application pool by way of a **Universal Resource Identifier** (**URI**) namespace. Application pools will be covered in detail later – for now, consider an application pool as one or more web applications on your server. Each application pool registers those portions of the URI namespace for which it services requests, and receives its own request queue within HTTP.sys. An application pool may be servicing more than just one portion of the URI namespace.

Since HTTP.sys runs completely within kernel mode, any problems in user mode don't affect it. Even if an application pool has to be restarted (for any reason), HTTP.sys will continue to queue requests for that application pool, anticipating that the pool will recycle and begin to accept requests again. When IIS shuts down, it removes its application pools and their URI namespace registrations from HTTP.sys. This way, while HTTP.sys continues to operate and serve applications other than IIS, it will no longer queue requests for the shutdown IIS application pools.

URI Cache

HTTP.sys implements a URI response cache, allowing it to serve cached responses completely within kernel mode and avoiding a costly transition to user mode. By avoiding the transition from kernel mode to user mode, literally thousands of CPU cycles are cut from each request, and the overall code path to serve a response from the cache is significantly shorter. The mechanism for transitioning to user mode relies on the **Windows IO Manager**, which must acquire, process, and complete IO Request Packets for the transition. This costly procedure is avoided completely when a response is served from the cache, and a performance gain on the order of 100% can be achieved. HTTP.sys has an advanced algorithm for determining what is placed in the cache, basing its decisions on the distribution of requests that a particular application receives.

Quality-of-Service

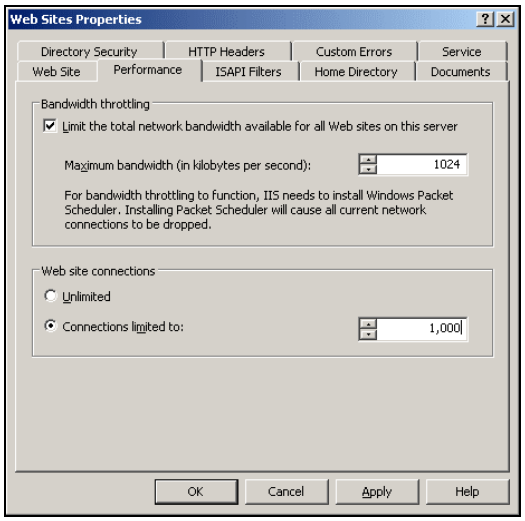
Quality-of-Service describes the methodology behind managing server resources like memory or CPU cycles. An administrator may control the resources allocated to sites and application pools, thereby affecting the quality of service that other sites and applications receive. Quality-of-Service components include:

- **Bandwidth Throttling**
- **Connection Limits and Timeouts**
- **Application Pool Queue Length Limits**

Bandwidth Throttling

Bandwidth Throttling allows you to limit the amount of bandwidth available to each individual web site. In the case of a server with multiple sites, it often makes sense to limit available bandwidth for non-critical sites and to ensure adequate bandwidth for important sites. As far back as IIS 4, bandwidth throttling could be done on both site and server levels. IIS 6.0 takes advantage of the bandwidth throttling support provided by NT QoS services. With the addition of the `MaxGlobalBandwidth` setting, it is now possible to throttle all sites (that do not have their own individual settings) collectively at a given rate. Sites may have more specific settings, as provided by its individual `MaxBandwidth` setting, which will exclude the site from the global limit.

The bandwidth settings for the server and each web site can be set using the property pages. Open the IIS Manager (enter inetmgr in the Run command window) and right-click the appropriate entry in the left tree structure. That will bring up a menu, where you can click the Properties option and then the Performance tab. That will bring up the following property page:



Using the Performance property page, you can configure the bandwidth settings for both the server as well as individual sites.

Connection Limits and Timeouts

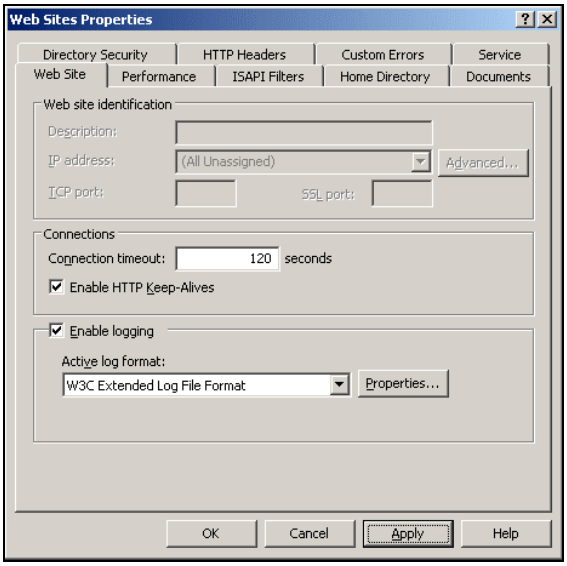
Connection Limits, as the name implies, limit the number of simultaneous connections at any one time. If a connection limit is reached, further attempts to connect will receive an error in response, and will then be disconnected. Limits may be set on a per-site basis as well as on the server as a whole. When a connection limit is reached, IIS 6.0 returns a 403.9 error code. In IIS 5, this error was customizable to be a file, URL, or the default text. In IIS 6.0, however, this error is not customizable, and returns the default text only. Connection limits are detected and acted upon from within `HTTP.sys`. The settings for connection limits can be set using the previous screen as shown.

IIS 6.0 supports four types of connection timeouts. The first three were available in IIS 5, and include a timeout where a connection has sent data but is now idle, a timeout where a connection has been established but no data has been sent, and a timeout on sending a response, based on a minimum bytes-per-second value. The fourth, new to IIS 6.0, is a timeout designed to prevent clients from sending data at an unacceptably slow rate. This `ReceiveEntityBody` timeout uses the `ConnectionTimeout` value to ensure that entity bodies are received in a timely manner. Once IIS knows that a request has an entity body, it starts a separate timer for receiving the entity body. This time is reset each time a packet of entity body data is received. If it times out, the connection is closed. The various timeouts provided by IIS 6.0 are:

- **`ConnectionTimeout`**
  It specifies the amount of time the server will wait before disconnecting an idle connection. This is similar to the IIS 5 `ServerListenTimeout` property. In some instances, applications that use port 80 for other tunneling protocols may wish to keep the connection open, even though it is idle. In such cases, this timeout must be increased.
- **`MinFileBytesPerSec`**
  It specifies the minimum net bandwidth to determine how long it should take to send a response. In cases where your server is on a link that may be slow at times, increasing this value will ensure that valid connections are not closed.
- **`HeaderWaitTimeout`**
  It specifies the number of seconds the server should wait for all HTTP headers to be received before disconnecting the client. This aids in avoiding a common denial-of-service attack that attempts to create the maximum number of open connections.

Note that increasing any timeout values can be dangerous, as it can open up your server to denial-of-service attacks by allowing abnormally long requests to sit idle, thereby consuming resources.

The `ConnectionTimeout` property can be set for a web site using the property pages. The following screen can be obtained using the same manner as described previously, and clicking on the Web Site tab:



Connection timeout may be changed on the Web Site properties panel as seen in the previous screen.

Application Pool Queue Length Limits

Application pool queue length limits are used to prevent too many requests from being queued and overwhelming the server. If a new request would exceed the queue length limit, the request is rejected by sending a 503 error response and closing the connection. In cases where a single web site becomes so busy with requests that other sites on the server are suffering, an administrator might consider lowering the queue length limit for the application that serves the requests for that site. This is to constrain the number of requests that the server will queue, thereby freeing resources for other sites.

`HTTP.sys` sets the queue length limit to a default of 3000. IIS 6.0 resets this value to 1000 as the default. In cases where this limit is reached, an administrator might be advised to first evaluate what applications are running such that they are unable to serve requests fast enough and therefore the queue grows past 1000 requests. If the machine's CPU is not at its maximum load, **Web Gardens** are a possible solution, especially when an application is causing a queue to back up due to request processing taking an abnormally long amount of time (perhaps doing a very complex database operation). In cases where the CPU is at its maximum load, and the application cannot be further optimized, it might be time to invest in more server hardware.

Logging

Text-based logging of HTTP requests is now handled by `HTTP.sys`, bringing performance and reliability gains to IIS 6.0. Because this logging is done at the kernel level, the worker processes need not worry about concurrency issues when writing to log files. However, the ODBC and custom logging modules are still handled by the worker processes. These methods often (especially in the case of ODBC) rely upon a database (such as Microsoft SQL Server) that handles concurrency issues; therefore, worker processes should not be impacted.

! If custom log modules or ODBC logging is used, kernel mode caching will be automatically disabled by IIS. This is done specifically to prevent the log from missing hits that are retrieved from the cache.

Logging Mode (file format)	Process
W3C Extended (as defined by the World Wide Web Consortium's Working Draft *WD-logfile-960323*, found at http://www.w3.org/pub/WWW/TR/WD-logfile.html)	`HTTP.sys`
IIS	`HTTP.sys`
NCSA	`HTTP.sys`
Centralized Binary Logging	`HTTP.sys`
ODBC	`w3wp.exe`
Custom	`w3wp.exe`

For more detailed information about logging, see *Chapter 6*.

The Web Administration Service

The **Web Administration Service (WAS)**, also known as the process manager, is a user mode component of W3SVC, responsible for process management and configuration. WAS works with the metabase to handle the configuration information passed to `HTTP.sys` and is used in the management of worker processes. WAS is also responsible for starting and managing the operation of worker processes, including monitoring the health of worker processes. We refer to the Web Administration Service as WAS as well as the process manager.

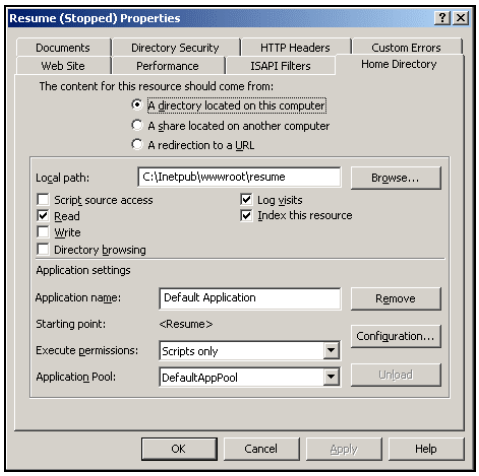
When IIS is first started, the WAS swings into action. It reads the configuration information from the metabase file, and initializes the namespace routing tables in `HTTP.sys`. Basically, it means that an entry is made for each application. This entry will help `HTTP.sys` to decide which application pool to forward the request, depending on the URLs mapped to the application pool. `HTTP.sys` uses this information to set up request queues for the application pools. All these steps are completed before `HTTP.sys` starts accepting requests.

When new applications and application pools are added, WAS configures `HTTP.sys` accordingly. This involves configuring `HTTP.sys` to accept requests for new URLs, setting up new request queues, and indicating which application pool to forward the requests for new URLs. WAS manages the lifetime of worker processes. That entails starting worker processes, monitoring their health, and restarting them as and when necessary.

Application Pools

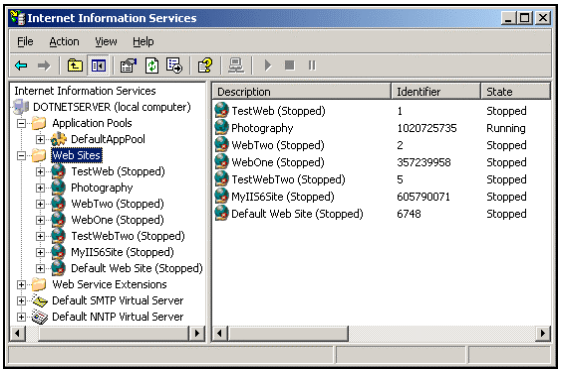
One of the key features of IIS 6.0 is Application Pools. Application pools define a set of one or more web applications served by one or more worker processes. Application pools allow different web applications to be separated such that they can be served by separate worker processes in separate application pools. Web applications can also be grouped into an application pool in order to share configuration settings. Each application pool is a separate Windows process (an instance of `w3wp.exe`), and is completely independent of other application pools, having no facilities for communicating between each other. Each application pool represents a request queue within `HTTP.sys`. They are considered completely segregated process spaces by design. Application pools can serve a single web application (such as an ISAPI application or ASP.NET page) or multiple applications. Multiple web sites may be placed in a single application pool, and any web directory or virtual directory can be assigned to an application pool.

The application and application pool for a web site can be configured by using the property pages for the web site. Clicking the Home Directory tab on the property pages displays the settings:



Application pools allow configuration settings to be specified independently from other groups of web applications. You can specify the health monitoring aspects of each application pool, and schedule application pool recycling based on the number of hits, and the amount of memory used, or uptime. Each application pool can be configured to conserve resources by stopping its worker process after a configurable amount of idle time, and limit the size of its request queue.

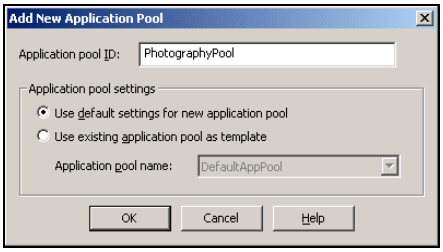
When IIS 6.0 is installed, one application pool is created as the default application pool for all sites. In the following screenshot, you will see that all of the web sites are running under the single default application pool; DefaultAppPool.



If you are hosting a single site, you should consider using this default application pool, though you may find it useful to rename it to something more meaningful, related to your application. If, on the other hand, you are hosting multiple web sites on a single server, you could create separate application pools for each site; thereby isolating them from interfering with each other, and allowing you to maintain stricter security control by configuring the worker processes to have different privileges.

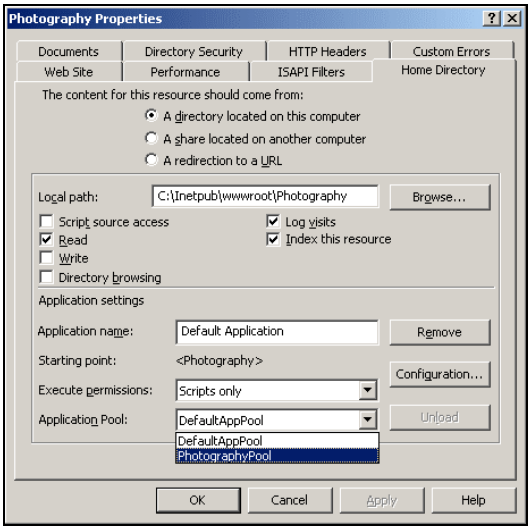
In the previous example, if we wanted to isolate our Photography site, perhaps because we wanted more control over the applications within the site, we could create a new application pool and place the site within it.

In the previous figure, right-click on the Application Pools, then click New | Application Pool to bring up the Add New Application Pool panel:



Creating a new application pool is as simple as providing it with a name, and giving it either default settings (which you can customize later), or copying the settings of an existing application pool.

Selecting the application pool for your application is as simple as choosing an available application pool from the drop-down box located in the Home Directory tab of your application's properties:



In this scenario, the Photography web site was put into its own application pool, isolating it from other applications. In a real-world scenario, this could be done because there might be a large number of sizable images being displayed. Hence, finer control over configuration will be desired, as well as a higher degree of isolation for image processing applications.

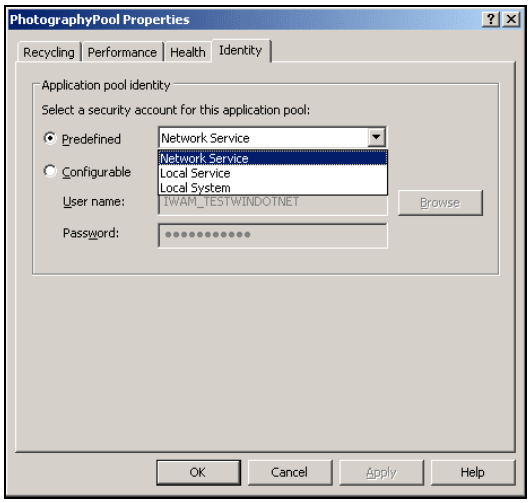
Another possible scenario in which application pools are useful is where you have both a staging and a production version of the same site. You might consider using different application pools to provide more separation (and possibly different configurations) between the production and testing versions.

Separating applications into independent processes eases a number of management tasks, such as bringing a site online or offline, making changes, managing resources, or debugging. This is definitely an advantage over previous versions. Much like the architecture of IIS 5 out-of-process applications (like ISAPI applications), separation is determined by URI namespace. As described before, `HTTP.sys` routes requests to a particular application pool based on a combination of either web site name or IP address, port, and URL prefix.

Application Pool Identity

The identity of an application pool is the **user account** under which the worker process runs.

You can assign a pre-defined account or a user-configurable account to an application pool as its identity. Changing the identity is accomplished from the application pool properties page; accessible by right-clicking on the application pool you wish to change:



The identity is configurable via an application pool property, represented in the metabase as `/LM/W3SVC/AppPools/<AppPoolID>/AppPoolIdentity`, and contains the following possible values:

Property Value	Description
0	**`LocalSystem` Account**. Member of the `IIS_WPG` group. The `IIS_WPG` group is a user group installed by IIS 6.0 that provides the minimum set of privileges required by IIS. This group provides a convenient way to use a specific user for the identity account without having to manually assign the proper privileges to that account. If the configured account you create is not in the `IIS_WPG` group and does not have the appropriate permissions, the worker process will not start, and an error will be logged to the system event log.
1	**`LocalService` Account**. Member of `IIS_WPG` group. Unlike `NetworkService` (below), the `LocalService` account has no network privileges, and should be used if the web application has no need for access outside the server upon which it is hosted.
2 (default)	**`NetworkService` Account**. Member of `IIS_WPG` group. This is the lowest privileged account of the three pre-defined accounts.
3	**Configured Account**. Set the property `WAMUserName` and `WAMUserPass` to the name and password of the account to use.

It is always good practice, when choosing an identity, to select the least possible privileges necessary to accomplish your goals. An identity with privileges like `LocalSystem` will give your application permissions that might constitute security vulnerability, should your application be compromised.

See also the discussion on impersonation in *Chapter 7*.

Demand Start of Application Pools

When the first request for a URL arrives, and if it is a part of the namespace for an application pool, the worker process (or first worker process in the case of a Web Garden) for that application pool is started. This feature, known as **Demand Start**, ensures that processes for little–used applications aren't started, even when there are no requests, and would consume resources by sitting idle.

The process manager reads the configuration information for application pools at startup time. The configuration manager initializes the namespace routing table within `HTTP.sys`, with one entry for each application pool. These entries comprise the regions of URI namespace for each application pool and the maximum number of worker processes for the group (typically only one, but more than one in the case of Web Gardens). This initialization configures `HTTP.sys` to recognize that there is an application pool available to respond to requests in a particular part of the namespace. When a request comes into `HTTP.sys` for an application pool and there are no processes available to handle it, `HTTP.sys` notifies the process manager, which starts the worker process. In the interim, `HTTP.sys` queues the request (and any further requests) until the process is ready.

Application Health Monitoring

Nobody writes perfect software. Even the best applications have bugs. IIS 6.0 helps manage this inevitability by constantly monitoring the health of applications and taking both preventive, as well as corrective, measures. IIS 6.0 can help diagnose problems such as memory leaks over time and access violations, and help take appropriate actions to deal with them. WAS will consider an application "unhealthy" if it has crashed, hung or terminated abnormally. All of the available IIS threads in the worker process are blocked, or the application notifies IIS directly that a problem exists.

Health Detection

You can configure an application pool to periodically "ping" a worker process, and take action if a response is not received. **Pinging** is accomplished via a named pipe between WAS and the worker process. A message is sent between WAS and the worker process over the named pipe. If the ping succeeds (the message is received and a response is sent), WAS will presume that the worker process is in good health. If there is a problem, like the worker process not responding in time, WAS can either restart the worker process or execute user-defined actions. Worker processes implement pinging in their own thread pool, so you need not implement any special code in your applications to benefit from this feature. The `PingingEnabled` metabase property controls whether an application pool will implement pinging, and the `PingInterval` property configures the number of seconds between successive checks.

Normal operation of an application

When a worker process is performing normally, it sends `ReceiveRequest` message to `HTTP.sys`. Once `HTTP.sys` receives a request, it will try to complete a `ReceiveRequest` call from the worker process. If there are no such calls pending, the new request is queued in `HTTP.sys`, up to the queue limit.

When an application crashes

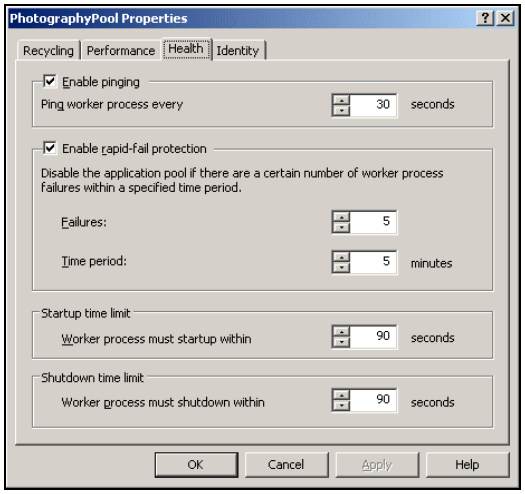
When a worker process crashes (and exits), any `ReceiveRequests` which have completed to the worker process, will have their respective connections reset. The client will see TCP resets (`Winsock 10054` errors). Any requests that are on the `HTTP.sys` queue are not negatively affected by a worker process crash, since they aren't associated with a worker process yet. When the process fails, the process manager will detect the crash and can restart the process, which will then process the requests on the queue.

> **This changes if a debugger (such as Visual Studio) is attached to the worker process. In this case, the debugger catches the crash. Since the process is still active, the client will see a hung connection and will eventually timeout. For details on how to attach a worker process to a debugger, refer to *Chapter* 7.**

Rapid Fail

In the event an application pool experiences a certain number of crashes in a certain amount of time, the pool may be completely shutdown. The number of crashes and the amount of time after which the pool is shut down can be configured for each pool. Additionally, you can manually put an application pool into a **Rapid Fail** state. When this state is entered, `HTTP.sys` will return a 503 Service Unavailable message to any requests for the shutdown application pool. This state reduces the processing overhead on the server, as requests for failed applications never make it out of kernel mode.

You can configure the properties related to the health of applications in a pool using the property pages of the application pool. Right-click on any application pool entry in the IIS Manager and select the Health tab:



Orphan Worker Processes and Debugging

If a worker process does not respond to a ping, the process still exists even though it is seen to be in a locked state. You can configure IIS to start an application or a script in this case. Additionally, if applications have the debugging action enabled, a new worker process will be started, and the misbehaving worker process will remain available for debugging. In other words, the worker processes will not be terminated by WAS, enabling you to attach a debugger for further evaluation.

! Leaving a large number of locked worker processes around without shutting them down can quickly eat up resources. If you enable debugging, make sure that you actually debug and then shut down the misbehaving process.

Applications declaring "Unhealthy"

ISAPI applications may directly report themselves as "unhealthy", so that they are recycled, via the new server support function `HSE_REQ_REPORT_UNHEALTHY`. To use this function, the application pool containing your ISAPI application must have pinging enabled, as it is during the ping that the unhealthy state is checked.

This is how ASP and ASP.NET recycle themselves. If the ASP ISAPI extension detects that too many of its threads are blocking, it will use this function to signal a recycle.

! Signaling that your ISAPI application is unhealthy and should be recycled will recycle the worker process for the application pool containing your ISAPI application. If there are other applications in the application pool served by that worker process, they will be affected.

Application Recycling

IIS can restart (refresh) worker processes within an application pool on a scheduled basis. This is especially useful in cases where you're running web applications that are both poorly behaved and cannot be modified. Recycling can be configured for elapsed time, number of requests served, specific times, memory usage, or on demand. Recycling happens in one of two possible ways; either the worker processes can be stopped and a new one started, or worker processes can be recycled in an "overlapping" fashion.

The first option is straightforward; worker processes are stopped when a certain condition that has been configured is satisfied for example, after the process services a certain number of requests. In an **overlapping recycle**, the running process is allowed to complete processing the remaining requests in its queue while a new process is created. Because `HTTP.sys` is responsible for queuing requests to application pools, a recycled worker process will have new requests held by `HTTP.sys` until it is ready to accept them, providing uninterrupted service. After the new worker process starts and is ready to accept traffic, new requests (as well as queued ones from the intervening time) are routed to the new worker process, while the old process finishes and shuts down gracefully. This provides uninterrupted service by allowing the old process to complete (or "drain") existing requests before it shuts down. If the old process takes too long (in the case where it has crashed and is hanging), IIS will shut it down forcibly.

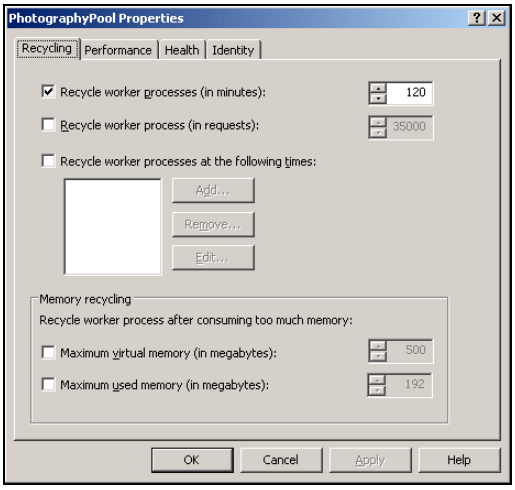
! Since the timeout value of an overlapping recycle (and for that matter, any shutdown condition) is configurable, it is possible to forcefully shut down the old process before it has finished servicing existing requests.

The following table outlines the different methods of process recycling. Note that these options are not mutually exclusive and you can use one or more of them at the same time. Remember, as far as possible, you should attempt to fix the problem at the source (your application) rather than rely on recycling.

Recycle Mode	Description
Elapsed Time	Recycles worker processes after an elapsed number of minutes. Use this mode if you know that your applications are failing after a certain time period.
Number of Requests	Recycles worker processes after a specific number of HTTP requests. Use this mode if you know that your applications are failing after a certain number of requests.
Scheduled Time(s)	Recycles worker processes at specified times within a 24-hour period.
Memory Usage	Recycles worker processes based on the amount of virtual memory used by the `w3wp.exe` process. Use this mode if you know that your application has a memory leak.
On Demand	Recycles worker processes when an IIS administrator instructs IIS to do so. Use this mode to recycle a particular application rather than restart the entire web service.

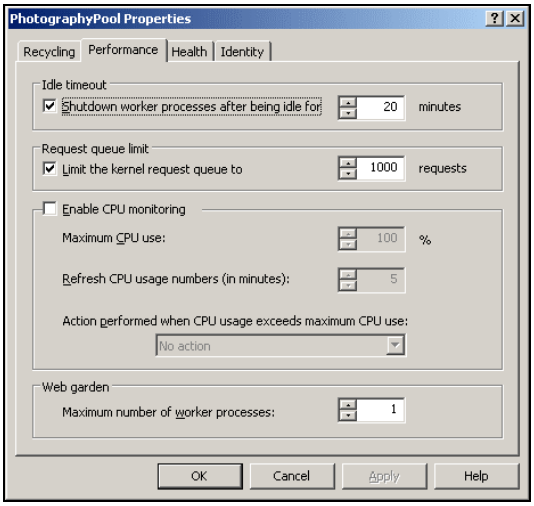
! When an application is recycled, you will almost surely lose any session state that you had stored, unless you have persisted it elsewhere (such as to a database).

You can set an application to be recycled by configuring the settings accessed through the properties page of an application as described for the previous screen and clicking on the Recycling tab:



Idle Process Timeout

An application pool can be configured to shut down worker processes that have sat idle for a configurable amount of time, thereby freeing up resources. New worker processes will be demand started when new requests are received for their application pool. The settings for timeouts can be accessed by opening the property pages for an application pool and clicking the Performance tab:



Web Gardens

An application pool may configure the amount of worker processes assigned to the pool. A **Web Garden** is created when multiple worker processes exist within a single application pool. Do not confuse Web Garden with Web Farm, which is a term used to describe multiple physical servers, all hosting the same web application (usually for load-balancing purposes). Web gardens improve both the performance and reliability of a web application. Web gardens can provide additional available worker processes, when one or more are heavily loaded or locked in the process of servicing a request.

Using Web Gardens

To make the best use of Web Gardens for process balancing, multiple application request queues are created and `HTTP.sys` is responsible for distributing the load across the processes in a Web Garden. `HTTP.sys` will route requests from different connections to the processes in a round-robin fashion. For example, if a Web Garden has three processes, the first three requests from different connections will go to the first, second, and third process in order. The fourth request will then go to the first process, and the cycle will continue. Considering the overhead involved in operations that are serialized for locking, this method provides performance advantages.

Session State in Web Gardens

Remember that HTTP is a stateless protocol. What that means is that if a connection is closed after processing a request, there is no guarantee that the same worker process will service future requests, even from the same client. In cases where session state must be maintained across different connections, the session state could be lost if a subsequent request were sent to a different worker process. While keeping the connection open is one possible solution, it is recommended that state be persisted to ensure that it is available. What this means is storing the state in an external location such as a database – for example, ASP.NET provides the ability to store session state in a SQL Server database.

Application Pool Parameters in Web Gardens

While most settings for application pools remain unchanged when operating a Web Garden, there are a few settings which have modified meanings. They are:

- **`MaxProcesses`**
  This is the property to change to create a Web Garden. By default, all application pools have one process and `MaxProcesses` is set to `"1"`. Increasing the value of `MaxProcesses` to the number of worker process instances, you will create a Web Garden for your application pool.

- **IdleTimeout**
  It is individually calculated on a per-process basis, allowing processes to timeout individually.

- **PeriodicRestartTime**
  It states that all of the processes in a Web Garden will recycle within the time specified. For example, if a Web Garden consists of four processes and `PeriodicRestartTime` is set to 16 hours, each process will be recycled every four hours. It should be apparent that this distribution of restart times maintains that each process is recycled every `PeriodicRestartTime` hours. In the event of a restart due to a crash or other problem, that time period is reset to when the process is restarted.

- **PeriodicRestartRequests**
  It operates in a similar fashion to `PeriodicRestartTime`. For example, in a Web Garden with four processes and a `PeriodicRestartRequests` value of 80,000 requests, the first process will recycle after 20,000, the second after 40,000, the third after 60,000, and the fourth after 80,000 requests. Each worker process will then recycle every 80,000 requests in this manner.

- **RapidFailProtection**
  It calculates the total number of failures across all processes in the Web Garden when comparing them to the time interval (`RapidFailProtectionMaxCrashes` over `RapidFailProtectionInterval`).

Processor Affinity

Processor affinity is an application pool property that forces worker processes to run on specific processors on a multi-processor machine. A Web Garden provides its best performance gains on a multi-processor server, where each worker process in a Web Garden may be assigned to run on a separate processor. For example, on a server that has eight processors, an application pool might be configured to run on four of those processors.

`SMPProcessorAffinityMask` is the mask for all processes running in a Web Garden. To set processor affinity, you must set the `SMPAffinitized` property to true, and then set the `SMPProcessorAffinityMask` to the range of CPUs you wish your worker processes to be bound. In the previous example, the affinity mask would be set for the processors numbered zero through three, and all worker processes in the application pool would run on the first four processors.

Summary

With the new kernel mode driver, `HTTP.sys`, and the new worker process isolation mode, IIS 6.0 is a platform that extends the capabilities as well as the performance and reliability of the web application platform. The new capabilities of IIS 6.0 allow for the offering of new services based on the web server, and the performance enhancements allow for more services to be offered on existing hardware, while simultaneously improving the reliability of those new as well as existing services. Coupled with an enhanced ability to maintain and configure these new features, the new request architecture of IIS 6.0 is clearly a step forward.

In this chapter, we looked at the new request architecture of IIS 6.0, which provides many improvements over the previous versions. IIS 6.0 separates the request processing code and the application handling code into `HTTP.sys` and Web Administration Service respectively. `HTTP.sys` is a kernel mode driver and provides facilities of kernel level request queues, response caching, logging, and configuring the quality of service provided to the clients. WAS uses the metabase settings to configure the request queues for `HTTP.sys`. It also provides the facility of creating and monitoring worker processes for applications to run.

IIS 6.0 provides a new Worker Process Isolation Mode, in addition to the IIS 5 Isolation Mode. The new mode entails more features, such as application isolation, multiple application pools, application health monitoring, rapid fail protection, and application recycling. Configuring an application pool as a Web Garden and using the processor affinity feature can improve the performance of the applications. IIS also provides the option of running applications in the IIS 5 isolation mode for backward compatibility in situations that require it. We saw how to choose between these two options and to switch between them.

To summarize, here are some of the features and components of the new request architecture:

- Kernel-mode HTTP listener (`HTTP.sys`)
- Web Administration Service (`WAS`)
- Application Pools
- Worker Process Isolation Mode
- Web Gardens
- Health Monitoring and Rapid Fail Detection
- Recycling
- Idle Timeout

In the next chapter, we will take a comprehensive look at the security aspects of IIS 6.0.