Making WMI Queries In C++

codeproject.com/Articles/10539/Making-WMI-Queries-In-C

Martin Friedrich May 31, 2005

Abstract

When searching the internet for documentation about WMI consumer programming, most of the documentation found is written for the C# programmer. Articles on using WMI from C++ are quite scarce. This is even more astounding as MSDN offers a lot of C++ code snippets. However, these snippets fail to constitute a real tutorial. This article deals with various aspects of how to implement WMI consumers. During the course of the accompanying examples, emphasis is placed on how to retrieve information from WMI by queries.

Despite its introductory nature, the article assumes the reader to be familiar with (D)COM programming in a C++-context. Some knowledge about SQL is an advantage, but not a requirement.

I Introduction

Motivation

Although the average user does not necessarily perceive it that way, operating systems are subject to an evolution that is as amazing as that of computer hardware. During the Stone Age, when dinosaurs ruled the earth, computers came as monstrous black boxes filled with wires and vacuum tubes, hardwired for certain tasks they were designed to perform. Such were the early ancestors of modern operating systems. In particular, they offered no support for administrative tasks. This was partly because at that time, with an installed user base of five machines, there was no need for such things.

Today, looking back from the preliminary endpoint of this evolution, things have changed drastically. Modern computers have become omni-present, all-around tools of unprecedented versatility. Their operating systems have become bigger and evermore sophisticated environments. Most importantly, they are not the "specialized architecture" machines they once were. Each machine can support an immense number of hardware and software configurations. Operating Systems reflect this; on the Windows platform, for example, starting with Windows 95 Microsoft introduced the registry as a centralized place to store configuration data. Other tools and components for system management were added.

Because of networking capabilities being a key requirement nowadays, integrated operating system support to administrate not only the local computer workstation at your desktop, but the networking infrastructure as well, is of ever growing importance. With company networks easily consisting of 100+ computers, remote administration capabilities are of particular advantage. WMI (Windows Management Instrumentation) is the integrated solution to these needs.

A prominent example of a WMI application is the built-in task monitor in Windows XP and Windows 2000. It impressively demonstrates an important feature of WMI - there is an API for WMI which can be called from your own applications. This means that the rather cumbersome way of calling utility applications, e.g. *PING.EXE*, via the command line interpreter and parsing its output for results is not needed anymore.

From the programmer's perspective, WMI is a DCOM based service that provides various methods of synchronous and asynchronous access to many system management related data structures. Although much of this information may be accessed by other means, it is the unifying API that can make WMI quite benign to the programmer.

The remainder of this section will give a short overview of WMI and WQL, its customary SQL derivate. The second section deals with synchronous queries on the local computer. Asynchronous queries are the topic of section three. Finally, in the fourth section, a third kind of communication with WMI, events, are discussed.

Overview of WMI and WQL

Windows Management Instrumentation or WMI provides status or performance data about computers - possibly in networks that might even encompass whole enterprises - to applications. It retrieves this data from various sources like the Win32 (resp. Win64) system, the registry or other, custom defined sources. In particular, the actual location of data is hidden by the WMI-API in favor of a more abstract view of the entirety of status or performance data.

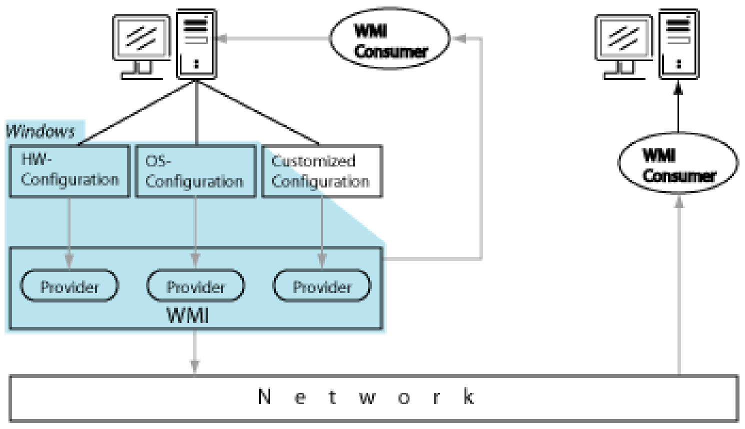

Figure 1

WMI introduces the notion of a *managed object*. A managed object is some logical or physical component on a computer that is managed by WMI. For example, a partition on a hard disk drive is a managed object. Another example would be a SNMP service or a physical component like the processor. On a more elevated level, say, for a network, router configuration could be managed by WMI. Applications that obtain information about managed objects from WMI are called *consumers*.

To a consumer, managed objects are represented by *providers*. A WMI provider is a COM component, that maintains information from a managed object and relays it to the consumer via WMI. Providers implement specific interfaces defined by CIM and are registered with WMI. By implementing these interfaces, developers can write their own providers specific to their needs. WMI comes with a number of predefined providers, which are called *standard providers*. The examples throughout this article use the standard providers only. Figure 1 illustrates this; interaction with providers is blurred by WMI, to the client both standard and custom providers are hidden; data structures exposed by WMI do seem to have merged with the operating system.

Information about managed objects is exposed by WMI as instances of classes created by WMI providers. In general, consumers do not access these instances directly, however. Instead, consumers *query* WMI, which then relays the query to the appropriate provider. From the requested subset of the managed object's properties, the provider builds a result set which is returned to the consumer, again via WMI (Figure 2). Queries are specified in *WQL*, a subset of ANSI SQL with a few extensions to fit WMI's needs. The most important difference between SQL and WQL is that WQL supports read-only queries only.

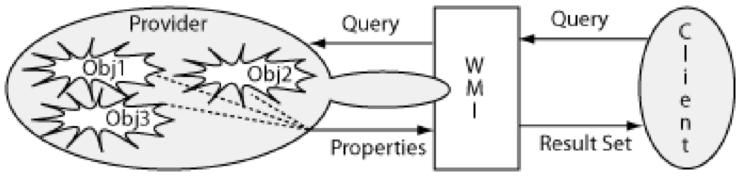

Figure 2

For someone familiar with SQL, WQL can be understood quite intuitively: To query a property of an instance, the property name (defined by the class of the instance) is inserted as a column name into a `SELECT` statement with the class name as table name. For example:

SQL

```
SELECT Name FROM Win32_Processor
```

yields the CPU name. As with SQL, an asterisk is a wildcard, meaning all columns are to be returned.

Depending on the mode of a query, its results are returned in result sets very similar to those of ADO or OLEDB. There are three modes to execute a query: Synchronous, asynchronous and semisynchronous. While the first two are exactly what their names suggest, the last one is of a hybrid nature: The query returns immediately with a valid result set, but the set's contents, i.e. the "real" result, are filled in later by WMI in an asynchronous manner. Selecting which query mode to be used has implications on performance and security.

A more thorough discussion of WQL and its various aspects can be found at MSDN Library.

II Synchronous Queries

COM and WBEM Locator Setup

As with any process or thread calling COM based services, the first step in writing a WMI consumer is to setup COM by calling `CoInitializeEx`. Usually - and absolutely necessary with asynchronous queries - for WMI `COINIT_MULTITHREADED` is specified. This puts the calling thread resp. process into a multi-threaded apartment. The exception are OLE clients - they are limited to apartment-threaded execution. This introduces a problem with asynchronous queries that is addressed in section II.

Unless an appropriate system-wide default security context is set in the registry, the next step is to set it process-wide by calling `CoInitializeSecurity`; as consumers only are considered here, `RPC_C_AUTHN_LEVEL_DEFAULT` and `RPC_C_IMPL_LEVEL_IMPERSONATE` are the only parameters to be set. Alternatively, or if `CoInitializeSecurity` has been called before with inappropriate values, `CoSetProxyBlanket` resp. `IClientSecurity::SetBlanket` can be called for a per-proxy setup.

After COM and COM-security have been set up, class `IWbemLocator` is used to access the WMI service:

```
CComPtr< IWbemLocator > locator;
HRESULT hr = CoCreateInstance( CLSID_WbemAdministrativeLocator, NULL,
                               CLSCTX_INPROC_SERVER, IID_IWbemLocator,
                               reinterpret_cast< void*** >( &locator ) );
```

`IWbemLocator::ConnectServer` connects to the specified host for WMI. For the purposes of this article the WMI service on the local machine is used.

`IWbemLocator::ConnectServer` is defined as follows:

```
HRESULT ConnectServer( const BSTR strNetworkResource, const BSTR strUser,
                       const BSTR strPassword,const BSTR strLocale,
                       LONG lSecurityFlags, const BSTR strAuthority,
                       IWbemContext* pCtx, IWbemServices** ppNamespace )
```

The first parameter, `strNetworkResource` specifies the CIM namespace the consumer is to connect to. Just like their C++ counterparts, CIM namespaces are used to disambiguate names in CIM hierarchies and impose some lexical structure over them. They take the form *\\server\namespace1\namespace2...* or *//server/namespace1/namespace2...*. `server` specifies the machine on which the namespace resides. If a namespace is located on the local machine, `server` can be omitted, i.e. namespace specifications takes the form *namespace1/namespace2...* or *namespace1\namespace2...*. `root\default` specifies the default namespace, which is always defined.

User and password are passed in the second and third parameters. For the local machine, they must be set to `NULL` or the call will fail. `strLocale` gives the locale with `NULL` denoting the current one. `lSecurityFlags` is normally set to `WBEM_FLAG_USE_MAX_WAIT`, which guarantees the call to return within two minutes. This safeguards consumers against blocking indefinitely if the targeted machine is down, for example.

In `strAuthority`, a Windows domain can be specified in which the user is to be authenticated. Again, this must be `NULL` on local machines or when specifying the domain in the `strUser` parameter. `pCtx` finally is set to `NULL`, except for rare cases when providers need an `IWbemContext` instance.

The namespace for most of the pre-defined WMI structures reside in the `root\CIMV2` namespace. For the local machine, the connection is established by:

```
CComPtr< IWbemServices > service;
hr = locator->ConnectServer( L"root\\cimv2", NULL, NULL, NULL,
                             WBEM_FLAG_CONNECT_USE_MAX_WAIT, NULL, NULL, &service );
```

When dealing with WMI, it is important to keep in mind that WMI APIs do expect `wchar` based strings.

Simple Queries - Querying CPU information (Project CPUTest)

This subsection covers the most simple scenario of using WMI - local synchronous queries without input parameters. As an example, the name of the local CPU and its clock frequency is to be found out. This information is held by WMI in instances of class `Win32_Processor`. For details of the various WMI classes - especially of which properties they own - see MSDN Library.

After successful service setup as described in the previous subsection, WMI can now be queried about the desired information. As this section deals with synchronous queries, `IWbemServices::ExecQuery` is called. Its prototype is:

```
HRESULT ExecQuery( const BSTR strQueryLanguage, const BSTR strQuery, LONG lFlags,
                   IWbemContext* pCtx, IEnumWbemClassObject** ppEnum )
```

The first parameter, `strQueryLanguage`, specifies the query language and must always be set to `L"WQL"`. The WQL statement for the query itself is passed in `strQuery`. `lFlags` specifies various flags, the most important is `WBEM_FLAG_FORWARD_ONLY`, which tells WMI to generate an enumeration as result set that can be traversed only once. Such enumerations are somewhat more efficient. Later on, other flags will be discussed as they become relevant. `pCtx` has the same meaning as before. The result set is returned as a pointer to an enumeration in `ppEnum`. For example:

```
CComPtr< IEnumWbemClassObject > enumerator;
hr = service->ExecQuery( L"WQL", L"SELECT * FROM Win32_Processor",
                         WBEM_FLAG_FORWARD_ONLY, NULL, &enumerator );
```

queries for all properties of instances of class `Win32_Processor`. The result set is returned in `enumerator`.

The result-containing enumeration can be traversed by calling `IEnumWbemClassObject::Next` repetitively. This method can return one or more `IWbemClassObject`-instances. As mentioned before, these instances are not the actual WMI managed object instances. In that case that would be instances of class `Win32_Processor`, they act instead as wrappers implementing access to the properties requested - and only those - by the query (Figure 3). However, they can be identified with each other and during the rest of the article this is done where it does no harm.

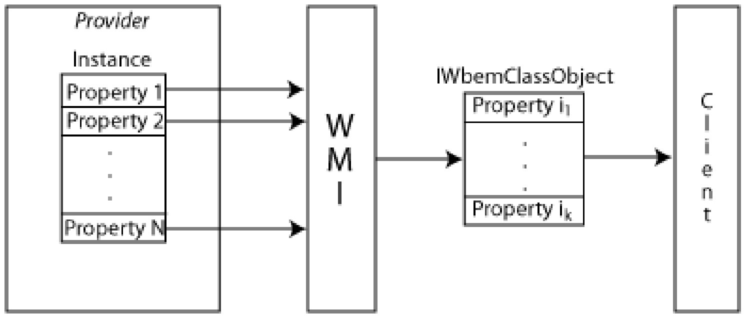

Figure 3

The number of instances returned in the enumeration is the number of instances of the class specified in the query, i.e. the argument to the `FROM`-clause. Some WMI classes are instantiated only once. Others may or may not be instantiated multiple times. `Win32_Processor`, for example, is instantiated for each CPU installed on the WMI host; that is, on single processor machines there is only one instance, on multi processor machines there may be more than one. For classes that are instantiated more than once, instances can be distinguished by their properties. This is also true, if a superclass name is the argument to the `FROM`-clause. How this is done will be discussed in the next subsection.

With regard to this, an important note to make here is that when dealing with WMI, `IWbemClassObject` is the only interface used by a consumer to access a managed object's data. When writing WMI providers, things will look substantially different, of course.

`IEnumWbemClassObject::Next` has the prototype:

```
HRESULT Next( LONG lTimeout, ULONG uCount, IWbemClassObject** ppObjects,
              ULONG* puReturned )
```

`lTimeout` specifies a timeout in milliseconds to wait for results in the result set to become available. If the timeout expires without results being available, `WBEM_S_TIMEDOUT` is returned. `WBEM_INFINITE` means an indefinite timeout. For synchronous queries, this parameter is effectively ignored as results - if any - are always available immediately. The next parameter, `uCount` is the number of objects expected to be returned at maximum in `ppObjects`. `puReturned` is the number of objects actually returned.

```
ULONG retcnt;
CComPtr< IWbemClassObject > processor;
hr = enumerator->Next( WBEM_INFINITE, 1L, &processor, &retcnt );
```

thus returns the first instance of `Win32_Processor` in `processor`.

Because only one instance of `IWbemClassObject` is extracted from `enumerator` in this example, `CComPtr<>` can be used for convenience. When extracting more than one instance by a call to `IEnumWbemClassObject::Next`, an array of plain `IWbemClassObject*` must be passed and `IUnknown::Release` called later on each pointer in the array.

Properties of the instance of class `Win32_Processor` queried are returned in `processor`, which will then be used to retrieve those of interest. This is done by calling `IWbemClassObject::Get` with the name of the property in question supplied:

```
HRESULT Get( LPCWSTR wszName, LONG lFlags, VARIANT* pVal, CIMTYPE* pvtType,
             LONG* plFavor )
```

The property value is returned in a `VARIANT` union in `pVal`. The `_variant_t` wrapper class can be used here. Along with the property's value, its CIM type and origin information can be returned. For example, the CPU's name is retrieved like this:

```
_variant_t var_val;
hr = obj->Get( L"Name", 0, &var_val, NULL, NULL );
```

Passing Arguments To Queries (Project LocalPing)

Evidently, the rather primitive methods of querying WMI presented so far are simply not sufficient. While there are just a small number of installed CPUs on a given machine (usually), instances of other managed objects may reside in greater numbers on a machine, representing e.g. administrative entities.

As soon as a computer is connected to any network, for example, information about networking infrastructure must be processed. This may include anything from high-level network services and tools to low-level protocol stack. WMI provides specific classes to access such information. One example is the well-known tool *PING.EXE*. It is used to determine the reachability of a given IP address on a network.

WMI provides built-in ping functionality by exposing the class `Win32_PingStatus`. Instead of building a command line for *PING.EXE* and executing it via one of the `exec` library functions, one can simply query WMI for an instance of `Win32_PingStatus`. Of course, in general one would rather like to ping very specific IP addresses instead of the whole internet. Therefore, the IP addresses to be pinged must be communicated to WMI by some means.

Target IP addresses are specified via the `Address` -property of class `Win32_PingStatus` . This is done by extending the `SELECT` -statement of the query with a `WHERE` -clause. A simple `WHERE` -clause takes one of these forms:

```
WHERE property operator constant
WHERE constant operator property
```

Lexically, this is very similar to SQL semantics. The main difference is that in SQL `WHERE` acts more as a filter while in WQL `WHERE` additionally can specify input arguments. The following snippet shows how to ping a given IP address, in this case the local loopback interface:

```
CComPtr< IEnumWbemClassObject > enumerator;
hr = service->ExecQuery( L"WQL", L"SELECT * FROM Win32_PingStatus " \
                         L"WHERE Address=\"127.0.0.1\"",
                         WBEM_FLAG_FORWARD_ONLY, NULL, &enumerator );
```

The result of the ping, i.e. whether the target IP can be reached or not, is returned in the `StatusCode` property. A value of `0` indicates success. Except for the different WMI class to be queried and the `WHERE` -clause, the associated source code in the "LocalPing" project works pretty much the same as the sample code from the previous subsection and thus does not need to be discussed further.

Multiple arguments to WMI can be specified by combining expressions with operators, like:

```
CComPtr< IEnumWbemClassObject > enumerator;
hr = service->ExecQuery( L"WQL", L"SELECT * FROM Win32_PingStatus " \
                         L"WHERE ( Address=\"127.0.0.1\" ) " \
                         L"OR ( Address=\"192.168.1.1\" )",
                         WBEM_FLAG_FORWARD_ONLY, NULL, &enumerator );
```

When retrieving the query's result, keep in mind that the enumeration returned now will contain more than one instance. For details of `WHERE` -clauses, refer to MSDN or the VS2003 online documentation.

Querying Multiple WMI Classes Simultaneously

Sometimes it is a good idea to execute two or more queries at once to achieve a better runtime behavior. The intuitive approach - specifying multiple classes in the `FROM` -clause as one would do in SQL - is not supported by WQL. The solution is to query for instances of base classes and then restrict the result by using the classes' *system properties*. WMI system properties are pseudo properties that exist for every class. Alongside other information, their purpose is to provide reflective information. For example, one can retrieve the class name on an instance by accessing the `__CLASS` property.

All classes of which instances are to be queried from within a single `SELECT` -statement must share at least one common base class, direct or indirect, which is specified in the `FROM` -clause. This implies that classes that do not share a base class cannot be queried this way. For example, instances of classes `Win32_Keyboard` and `Win32_PointingDevice` can be queried like this:

```
SELECT * FROM CIM_UserDevice WHERE ( __CLASS="Win32_PointingDevice" )
                             OR ( __CLASS="Win32_Keyboard" )
```

In general, the returned enumeration will be heterogeneous. Therefore care must be taken to use the correct property names, etc. on each instance. Again, the `_CLASS` property can be used to identify classes.

III Asynchronous and Remote Queries

Asynchronous Queries (Project AsyncPing)

So far, querying WMI has been done in a synchronous way. That is, a thread of a WMI consumer launches a query and then waits for the result to be propagated back. In particular, while waiting, the threads stop execution and resume only after query completion. While this clearly works and fulfills its promises, it has a distinct disadvantage - WMI queries can take some time to complete; time that might be spend by the querying thread by computing some other task, allowing for better resource utilization.

WMI provides for asynchronous queries. When making asynchronous queries, the querying thread does not wait for the query to complete. Instead, it provides class instances implementing special COM interfaces, called *sinks*. A sink implements specific methods that are called by the WMI provider on behalf of the consumer to process the results of the WMI query. This means that both the WMI provider and the consumer do out-of-process execution when making asynchronous queries. For the consumer, result computation is done out-of-process (just as it is with synchronous queries). For the provider, result processing is done out-of-process, because it is the provider that triggers processing, not the consumer anymore. This situation has some implications concerning security, which will be discussed in the next subsection.

When making asynchronous queries, no resulting `IWbemClassObject` instances are returned from the call immediately. Instead, an instance of class `IWbemObjectSink` is passed to `IWbemServices::ExecQueryAsync` . This is the prototype for `IWbemServices::ExecQueryAsync` :

```
HRESULT ExecQueryAsync( const BSTR strQueryLanguage,
                        const BSTR strQuery, long lFlags,
                        IWbemContext* pCtx,
                        IWbemObjectSink* pResponseHandler )
```

The `lFlags` parameter is used to control certain details of how the query is to be executed. For example, as with synchronous queries one can request bidirectional iterators to be returned. Also, WMI can be instructed to report intermittent status of the ongoing call to the sink. `pResponseHandler` is the sink instance that implements the actual result processing. API-wise, these are the two sole differences in both approaches. The code for making the `Win32_PingStatus` query asynchronously looks like this:

```
CComPtr< CPingSink > pPingSink = new CPingSink;
hr = service->ExecQueryAsync( L"WQL", L"SELECT * FROM Win32_PingStatus " \
                             L"WHERE Address=\"127.0.0.1\"", 0, NULL,
                             pPingSink );
```

`CPingSink` is the afore mentioned sink class. It implements the COM interface `IWbemObjectSink`. This interface defines two methods (additionally to those of `IUnknown`, of course):

```
HRESULT Indicate( long lObjectCount, IWbemClassObject** apObjArray );
HRESULT SetStatus( long lFlags, HRESULT hResult, BSTR strParam,
                   IWbemClassObject* pObjectParam );
```

`IWbemObjectSink::Indicate` is called to process query results returned in `apObjArray`. The number of results in this array is indicated by `lObjectCount`. `Indicate` may be called multiple times by the provider, if necessary. WMI providers call `SetStatus` to indicate completion and/or progress information to the consumer. If `WBEM_FLAG_SEND_STATUS` has been set on the `lFlags` parameter of `ExecQueryAsync`, `SetStatus` may be called multiple times by the provider. If not, it is called exactly once and indicates completion of the query. `strParam` and `pObjectParam` are used for complex error information in the former case.

Making Asynchronous Queries More Secure (Project SecAsyncPing)

The decision to interact with WMI asynchronously has two major implications. While synchronous queries are of plain, single-threaded nature, asynchronous queries - or rather the result processing involved - are more of a multi-threaded type: the querying thread, i.e. the consumer, continues after launching the query and some time later the WMI thread will execute the object sink's callbacks. Code controlling the lifetime of the object sink must reflect this.

The second implication is a direct corollary of the first - as WMI executes the object sink's callbacks in the process of the consumer, it might do this with a security context different from that of the consumer. The WMI provider with lower privileges may thus be allowed to access higher privileged consumers' data structures. To the consumer, it is a trusted component. Evidently, there are situations when this is not acceptable. One solution, discussed in the next subsection, is to use semisynchronous queries instead. The other one is to execute the object sink in an unsecured apartment.

Executing the sink in an unsecured apartment means executing it in another, special process that performs access checks. Depending on whether the consumer runs on Windows XP or Windows Server 2003 platform, the details of this do slightly differ from each other. The basic idea is to wrap around a stub object that is passed to `IWbemServices::ExecQueryAsync` instead of the sink object. The stub object relays the provider's calls to the actual sink, whose execution is hosted by *UNSECAPP.EXE*. Access checks are performed by the implementation of *UNSECAPP.EXE* in case of Windows Server 2003 or by the sink itself inside the `IWbemObjectSink::Indicate` and `IWbemObjectSink::SetStatus` methods. WMI setup and result processing by the sink object are just the same as in the subsection before.

To facilitate this approach, the WMI consumer instantiates the class `IUnsecureApartment` by calling `CoCreateInstance` :

```
CComPtr< IUnknown > pApartment;
HRESULT hr = CoCreateInstance( CLSID_UnsecuredApartment, NULL, CLSCTX_LOCAL_SERVER,
                               IID_IUnsecuredApartment,
                               reinterpret_cast< void** >( &pApartment ) );
```

Under Windows Server 2003, the returned instance is of class `IWbemUnsecuredApartment` , actually. Using this class instead of the also supported `IUnsecuredApartment` makes securing the callback somewhat easier. Of course, consumers running under Windows Server 2003 can use `IUnsecuredApartment` if they wish to. It should be noted here, that one does **not** instantiate `IWbemUnsecuredApartment` directly. MSDN is dead wrong in this example.

The actual stub creation is done quite easily. When running under Windows XP, consumers call `IUnsecuredApartment::CreateObjectStub` :

```
HRESULT CreateObjectStub( IUnknown* pObject, IUnknown** ppStub )
```

`pObject` is the sink object, `ppStub` is the newly created stub object. The method will return `E_POINTER` , if `pObject` is `NULL` . As mentioned before, implementation of access checks is the responsibility of the consumer. In the example project, this is class `CPingSinkUnsecure` .

Under Windows Server 2003, no access checks are done by *UNSECAPP.EXE* if the registry value under *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM\CIMOM\UnsecAppAccessControlDefault* is zero, the default value. Under pre-Windows Server 2003 platforms, this key is not available. When creating the stub object, this behavior can be overridden. Stub creation is done by calling `CreateSinkObject` on the `IWbemUnsecuredApartment` instance:

```
HRESULT CreateObjectSink( IUnknown* pSink, DOWRD dwFlags, LPCWSTR wszReserved,
                          IWbemObjectSink** ppStub )
```

Called with the original sink in `pSink`, the value of `dwFlags` controls the behavior of *UnsecApp.exe*: `WBEM_FLAG_UNSECAPP_DEFAULT_CHECK_ACCESS` uses the registry mentioned above, `WBEM_FLAG_UNSECAPP_CHECK_ACCESS` and `WBEM_FLAG_UNSECAPP_DONT_CHECK` do or do not perform access checks, ignoring the registry value. For example, to force access checks, one would call `CreateSinkStub` like this:

```
CComPtr< IWbemObjectSink > pStub;
CComPtr< IWbemUnsecuredApartment > aprt
         = pApartment->QueryInterface( IID_IWbemUnsecuredApartment,
                                       reinterpret_cast< void** >( &aprt ) );
HRESULT hr = aprt->CreateSinkStub( pSink, WBEM_FLAG_UNSECAPP_CHECK_ACCESS, NULL,
                                   &pStub );
```

Passing the stub instead of the sink object the consumer then makes the query just like before in the previous subsection:

```
hr = service->ExecQueryAsync( L"WQL", L"SELECT StatusCode FROM Win32_PingStatus " \
                                 L"WHERE Address=\'127.0.0.1\'", 0, NULL, pStub );
```

Semisynchronous Queries (Project SemiSyncPing)

As set out in the previous subsection, asynchronous queries may cause some security concerns because they assume the WMI provider to be a trusted component. There are scenarios when this is not acceptable. Instead of falling back to the rather inefficient approach of synchronous queries, WMI consumers can opt to make queries *semisynchronously*. This is an execution mode, which combines the secure handling of synchronous query results with the multi-threaded nature of result computation of asynchronous queries. At the same time, it eliminates the need to implement sink classes; results are returned by `IEnumWbemClassObject` instances directly, just as with synchronous queries.

The mechanism behind semisynchronous queries is therefore quite simple when compared to synchronous ones: A synchronous query computes its results (possibly in another thread or process) and returns only after completion with a prepared enumerator as its result, while a semisynchronous one prepares the enumerator, marking its elements as pending and then returns with the actual result computation continuing in another thread. In both cases, when calling `IEnumWbemClassObject:Next`, this method inspects the state of the next element in the enumeration, waiting for the specified time for the state of the element to become non-pending if necessary.

The accompanying source code reflects these similarities between the synchronous and semisynchronous mechanisms. Except for an additional `WBEM_FLAG_RETURN_IMMEDIATELY` flag in the `lFlags`, for semisynchronous queries the `ExecQuery` call is the same. This flag instructs WMI to handle the query semisynchronously and complete the `ExecQuery` call immediately with `WBEM_S_NO_ERROR` as result code (provided there are no other error conditions pending):

```
CComPtr< IEnumWbemClassObject > enumerator;
hr = service->ExecQuery( L"WQL", L"SELECT * FROM Win32_PingStatus " \
                         L"WHERE Address=\"127.0.0.1\"",
                         WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
                         NULL, &enumerator );
```

Upon successful completion of the call, the consumer now polls actively for results from the enumeration returned in `enumerator` by calling `IEnumWbemClassObject::Next` with a given timeout in the `lTimeOut` parameter:

```
while ( ( hr = enumerator->Next( 1L, 1L, &ping, &retcnt ) ) == WBEM_S_TIMEDOUT );
```

With synchronous queries, this is set to `WBEM_INFINITE`, indicating the timeout never to expire. `WBEM_NO_WAIT` means no timeout at all. If a result in the enumeration becomes available, the `Next` call will return with `WBEM_NO_ERROR` result code. Timeout expiration is indicated by returning `WBEM_S_TIMEDOUT`. If the result set is empty or its end has been reached, `WBEM_S_FALSE` is returned.

If the enumeration contains more than one element and the consumer determines it does not want to use all of them, calling `Release` on the enumeration will cause WMI to stop computing results and eventually free the enumeration and its contents.

Semisynchronous queries are the recommended way to make non-synchronous queries. They are easier to set up than asynchronous ones and they avoid the multi-threading and security issues associated with them. However, there may be situations requiring the asynchronous approach. For example, the requirement of consumers polling actively might be difficult to be implemented in certain scenarios.

IV Event Notification Queries

So far WMI has been instrumentalized for inspection of simple data representing managed objects. But there is more to come. WMI also allows for applications being notified if certain events are triggered. Requesting such notifications is done by making an *event notification query*. This special type of query differs from the ones seen before in two ways: First, event notification queries can be made asynchronous or semisynchronous, but never synchronous. Secondly, the programming model of event consumers is slightly different.

Permanent and Temporary Event Consumers

An application that is interested in receiving event notifications from WMI, is called an *event consumer*. Event consumers come in two types - *permanent* and *temporary* event consumers. A temporary event consumer is an application that wants event notifications for some span of time. After this span of time has been elapsed, typically limited by the application's lifetime, the application unregisters its query with WMI. In contrast, permanent event consumers will remain active beyond the application's completion. Windows XP comes with some permanent event consumers pre-installed, allowing to tie events to execution of command line scripts, for example.

These pre-installed consumers are known as *standard* event consumers. As both kinds of event consumers do not differ very much from a programmatic point of view, for the rest of this article permanent event consumers are left aside.

To register with WMI to receive event notification, an event consumer prepares a query in the form of a WQL statement. Within the WQL query, the type of event is specified by the `FROM` -clause. As usual, a base class can be used to specify multiple types. However, to constitute a valid query, the specified class must be derived from the class `__Event` . MSDN lists the possible (base) classes. Events relating to changes in the WMI data model are called *intrinsic*. Specifically, intrinsic events are generated by WMI itself; they do not need to have a provider present. Intrinsic events are limited to creation, modification and deletion of classes, instance and namespaces plus some of a more administrative nature.

Events that do reflect changes outside the WMI data model are called *extrinsic* and must be derived from `__ExtrinsicEvent` . They are only mentioned here for the sake of completeness.

WQL Queries for Events (Project DiskChange)

As said above, to request event notification, a consumer performs a WQL query with the event class specified in the `FROM` -clause. Additionally to the queries seen before, the `FROM` -clause is followed by a mandatory `WITHIN` -clause. This clause specifies a time interval in seconds, telling WMI how often to update the result containing enumeration. This is the major difference to the query types encountered in the previous sections - WMI keeps on sending event notifications to the consumer (via the enumeration or sink passed in the query call) until told to stop. This is done by calling `Release` on the enumeration or `IWbemServices::CancelAsyncCall` for the sink object for the asynchronous case.

There are only a handful of intrinsic events and they are of a more generic nature - creation, modification and deletion of classes or instances. Accordingly, they do not carry the pertinent information directly. Instead, they have a property referencing the actual data. For example, event classes referring to instance operations contain a property `TargetInstance` , which is inherited from `__InstanceOperationEvent` , that designates the involved instance. This property is checked in the `WHERE` -clause against the class one is interested in by use of the `ISA` operator.

Changes to logical disk drives - floppies, CD/DVD drives or hard disks - are indicated by `__InstanceModificationEvents` with instances of `Win32_LogicalDisk` as `TargetInstance` values. Requesting event notification of such changes can thus be done by the following WQL statement:

```
SELECT * FROM __InstanceModificationEvent WITHIN 10
         WHERE TargetInstance ISA 'Win32_LogicalDisk'
```

Here WMI is instructed to check every 10 seconds for new modifications of all available `Win32_LogicalDisk` instances and notify the event consumer accordingly. Such events are triggered by insertion of a CD or DVD into a drive, for example. Of course, such rather generic notification requests can be made more event-specific by restricting them to special instances:

```
SELECT * FROM __InstanceModificationEvent WITHIN 10
         WHERE ( TargetInstance ISA 'Win32_LogicalDisk' )
           AND ( TargetInstance.Name = "G:" )
```

This query limits its results to the logical drive 'G:'.

The event notification query is made not by calling `IWbemServices::ExecQuery` but by calling `IWbemServices::ExecNotificationQuery` or `IWbemServices::ExecNotificationQueryAsync` in the asynchronous case. In both cases, parameters are the same as with normal queries with the additional constraint that for `ExecNotificationQuery` flags `WBEM_FLAG_RETURN_IMMEDIATELY` and `WBEM_FLAG_FORWARD_ONLY` must be specified. If they are not, the call fails. As a consequence, calling `IEnumWbemClassObject::Reset` has no effect on such enumerations.

Retrieving the resulting events is done just exactly as before with semisynchronous queries (event notification queries are at least semisynchronous!). In addition, as long as `Release` is not called on the enumeration, it may receive further events.

Closing Remarks

This article did concentrate on the WMI consumer's point of view. However, many aspects remain uncovered, especially those of more direct interaction with managed objects did get their surface barely scratched. Traversing file system or directory managing structures and how to employ them to, e.g., copy files is an example. WMI does not only allow access to predefined data structures, it also provides meta-data capabilities to extend the existing and define new ones. All this is closely linked to the opposite role of WMI providers, and maybe another article from the WMI provider's side would be appropriate.

Caveats

As always, seemingly excellent systems come up with drawbacks if they are given a closer look. This holds for WMI, too, and indeed it can present some serious issues, which deserve a few words.

Firewalls and RPC

Being DCOM applications, WMI and WMI consumers will use RPC services when accessing objects remotely. Nowadays, this will almost certainly raise issues with firewall configurations. As a rule of thumb, TCP ports 135 and 445 will have to be exempted from the firewall's blocking list, together with ports allocated by DCOM dynamically. Minimizing this number of ports can be quite tricky.

When going across subnets in large, heterogeneously administrated corporate networks, this will prove a show stopper if sysops refuse to open the TCP 135 and 445 ports.

Remember to Remove All Event Notification Requests

When making event notification queries, remember to call `IUnknown::Release` on all `IEnumWbemClassObject` instances created by these queries. If this is not done, for example by exiting the program non-graciously via CTRL-C, WMI will continue to process these queries and though not returning results anymore, this will hog the CPU in an astonishingly fast and efficient way. See also KB327542 in the knowledge base.

WMI Queries Can Take Up Quite Some Time

Some types of queries can take a considerable amount of time to complete. Event notification queries, for example, may take a few seconds before delivering the first events, even if short polling intervals are specified. When designing an application, this should be addressed.

Asynchronous Queries and OLE

It seems that asynchronous queries do not like to be launched by apartment threaded threads, i.e. threads calling `CoInitializeEx` with `COINIT_APARTMENTTHREADED`. They exhibit the symptom that the `ExecQueryAsync` returns `WBEM_S_NO_ERROR`, but the sink is never called. The intuitive workaround would be to use `COINIT_MULTITHREADED` instead or make a semisynchronous query.

Sometimes using `COINIT_MULTITHREADED` is not possible, however, because the application needs to run in the single-threaded apartment. This is the case for OLE clients, for example. A solution to this situation would be to separate all the WMI consumer stuff into its own thread and initialize that one for free multi-threading.

Written By

Martin Friedrich
Web Developer

Germany

Still lacking an university degree in computer science, I have 20 years of experience in software development and implementation. Having expert knowledge in object-oriented programming languages like C++, Java and C# on Windows, LINUX and UNIX platforms, I participated in multiple research projects at the University of Oldenburg. During these assignments, I was trusted with implementation of a graphical editor for specification languages like CSP or Z and a prototypical tool for workflow data distribution and analysis. I gained experiences in a widespread spectrum of CS and software development topics, ranging from compiler construction across data base programming to MDA. My research interests include questions of graphical user interface design and component-based systems.