K2® Process-driven applications – fast.

.NET Best Practices Relevant to K2 Scalability

June 1

DOCUMENT VERSION | 1.0

CONTENTS

FUNDAMENTALS FOR .NET CODING IN K2 BLACKPEARL

- K2 hosts the standard .NET framework. It inherits its functionality, but also its limits. K2 does not (and cannot) circumvent standard .NET behavior or any "bad coding".
- K2 is constantly updated to look at and close any memory leaks; this does not absolve users of K2 from doing the same in their custom code.
- An assumption that *".NET has the Garbage Collector so memory will be fine*" is **<u>false</u>**.
- K2 may be running many instances of server events and other code items, so even "OK" code can behave as "bad" code in high load environments.
- K2 is a component in a solution along with SQL Server, Windows Server and others such as ASP.NET and SharePoint etc. All must be implemented on a scalable basis or K2 itself will not be (or seem) scalable. Problems in shared components are often exponential.
- Implementing .NET best practices in high load environments is a necessity for K2 scalability. <u>Every large scale K2 client around the World who has performance issues has been shown to have custom code with memory leaks or related issues.</u>

BACKGROUND

Understanding the .NET Garbage Collector (GC)

- The GC ***<u>attempts</u>*** *[1]* to clean up .NET objects in a timely manner with only a ***<u>limited effect on application performance</u>*** *[2]*

    [1] Not infallible
    [2] GC shortcuts have to exist for performance reasons
- BCL (Base Class Library; e.g. MSCORLIB, System.Data, etc.) team at Microsoft made assumptions of how objects should be used, such as the pooling of SQL Connections, to maximize GC behavior/performance.
- Inappropriate use of some objects can have severe memory and performance penalties.
- There's more to the GC than a success/fail for clean-up. Even objects that are ***<u>eventually</u>*** cleaned by the GC may be doing so highly inefficiently if they aren't used appropriately.

Finalizers

- Certain objects in .NET have native system objects as back-ends.
    - e.g. Sockets for network communication (used by SqlConnections).
- Finalizers are pieces of code ***<u>almost always</u>*** guaranteed to be called when an object is no longer needed, that ensure that native object handles are cleaned up.
    - "almost always" assumes process is disposed cleanly.
- Finalizers are stored and tracked in the Finalizer Queue.
- Keeping track of Finalizers is an expensive process. Objects should optimally be removed from the Finalizer queue if the developer destroys them.

- Avoids the GC wasting time traversing the queue (helping performance).
- Prevents timeouts during a Finalizer sweep – which would prevent it from cleaning up everything in the queue (creating memory leaks).

Generational Collection

- .NET GC is a "***Mark-and-Sweep***" "***Generational***" GC.
  - **Mark-and-Sweep** is an implementation detail and is not important.
  - **Generational** promotion is critical for .NET developers to understand, specifically how to prevent unnecessary generational promotion.
  - The GC keeps track of 3 generations. Generation 1 is where short-lived objects should typically 'live'. The GC always checks G1 first when it needs more memory. If an object 'survives' a G1 sweep it is promoted to G2 (and following that G3).
  - Short-lived (i.e. they ***should have been*** short lived) objects making their way to G3 frequently will cause colossal memory and performance issues.
  - An example is as follows. Arrows indicate ownership (e.g. Order object has an OrderLine property, OrderLine object has a Customer property):

    Order (G1) → OrderLine (G1) → Customer (G1) - *Collection Occurs. GC deems Order as not accessible (however, OrderLine and Customer are reachable even though their 'owner' is about to be cleaned up).*

    OrderLine (G2) → Customer (G2) **-** *After a large amount of G1 collections (typically on the order of 20-50) the GC does a G2 sweep. OrderLine can be cleaned up.*

    Customer (G3) - *Customer is probably never cleaned up. A G3 collection is an ultimate last-resort for the GC.*

    An object that could have been cleaned up immediately has been promoted to G3. The generational GC issue is compounded by Finalizers.

APPLICABLE .NET BEST PRACTICES

Best Practices – using block

- MS implemented ways to prevent performance penalties from finalizers and generational collection:

    - GC.SuppressFinalize – Indicates (to GC) that the object containing native references has been cleaned up and can skip finalization (i.e. removes it from the Finalizer Queue). ***You should never call this method unless you 'own' the object implementation.*** However, any object in the BCL that calls this method (e.g. SqlConnection) will support Disposal. Typically a disposable object looks like this:

```
class DisposableObject : IDisposable
{
  ~DisposableObject()
  {
    if(IsDisposed)
      return;
    IsDisposed = true;
    Dispose(false); // We pass false so that the universal dispose knows the GC is calling this code.
  }

  public void Dispose()
  {
    if(IsDisposed)
      return;
    IsDisposed = false;
    Dispose(true);
    GC.SuppressFinalize(this); // We cleaned up successfully; so there is no longer a need to call ~DisposableObject().
  }

  public void Dispose(bool disposing)
  {
    // Clean up native resources, e.g. socket handles.
    if (disposing)
    {
      // Clean up and null properties that are disposable themselves.
    }
  }
}
```

- What this means is that you can skip the entire penalty of Finalizers/Generational Collection by calling Dispose on an object when you are done with it. To add weight to the importance, Microsoft added a ***language shortcut*** in C# to deal with this:

```
using (SqlConnection connection = new SqlConnection())
{
  // Use the connection.
}
```

```
// The C# compiler turns this into the following code on your behalf.
SqlConnection connection = null;
try
{
  connection = new SqlConnection();
  // Your code.
}
finally
{
  if (connection != null)
    connection.Dispose();
}
```

- Always use Dispose on Disposable objects. If the object is only used for the duration of a method wrap it in a 'Using' block.
- Ensure that disposal is ***error safe*** (e.g. in Using block and/or well-structured try/catch)

Note:
You must ensure that SqlConnection, SqlCommand and SqlDataReader are all disposed (and error safe).

Best practices – Finalizer for Sqlconnection

- SqlConnections are very expensive to set up.
- In order to improve the performance characteristics of this object Microsoft decided to internally pool them (this implies that the GC will see them as referenced by 'something').
- Due to this if you are not disposing your SqlConnections correctly there is a very good chance the connection will stay open.
- This will permanently waste connections on your SQL Server and cause a myriad of problems for both K2 and other applications that might be using the same SQL Server.

Best practices – Avoiding "Straddling" connections

- Connections should be "atomic", meaning they should be doing a set of defined actions and then close the connection immediately. Otherwise:
  - Unnecessary multiple connections to SQL Server (and TCP/IP) will be made.
  - SQL Server threads will be taken up.
  - Deadlocks become a major risk (e.g. if same DB objects are used in both connections).
  - Unnecessary additional object references will be held in memory.
  - Exceptions/errors for one connection will cause a bubble up killing another connection, exposing memory leaks for both.
  - Object usage and garbage collection will be unpredictable.

- For example, the following should be avoided:

```
Connect to SQL Server
    Do something in SQL
    Connect to K2
        Do something in K2
    Close K2 Connection
    Do something in SQL
Close SQL Connection
```

Instead:

```
Connect to SQL Server
    Do something in SQL
Close SQL Connection
Connect to K2
    Do something in K2
Close K2 Connection
```

Best practices – Strings

- The following code is naïve and wasteful:

```
var myString = "Hello ";
myString = myString + "World" + someArg;
```

- This would allocate 2 unneeded strings (this is a simple example, but consider a dynamic SQL query); when only one result would be needed. Continually doing this will add an immense amount of strain to the GC – especially in a high load system.
- The reason this happens is strings in .NET are ***immutable*** – meaning you can never actually change the value of a string; you can only get a new one. The above example would have the 3 strings in memory (“Hello”, “Hello World”, “Final result”).
- To avoid this problem you can use several mechanisms:
    - **StringBuilder** – better for longer strings; or strings created during loops.

      ```
      var sb = new StringBuilder(); sb.Append("Hello "); sb.Append("World"); sb.Append(someArg); var myString = sb.ToString();
      ```

    - **String.Format** – better when you want to insert values into strings.

      ```
      var myString = string.Format("Hello {0}{1}", "World", someArg);
      ```

    - **String.Concat** – better for the above example.

      ```
      var myString = string.Concat("Hello ", "World", someArg);
      ```

As K2, just by cleaning up string usage in *SourceCode.Logging* we saw the K2HostServer idle memory usage drop from ~200MB to ~50MB.

Best practices – Events

- An event is a wrapper around a method pointer – part of a method pointer is also a reference to the target object. Events are highly risky for memory leaks. For example:

```
MyStaticClass.StaticEvent += myObject.Something;
myObject.DoSomething();
// End of method
```

- The static (rooted) object will now have a reference to myObject – which will disallow the GC from collecting myObject. Fixing this is easy, as indicated in the following code:

```
MyStaticClass.StaticEvent += myObject.Something;
try
{
  myObject.DoSomething();
}
finally
{
  MyStaticClass.StaticEvent -= myObject.Something;
}
```

- It is always a good idea to 'null' your events during disposal (if in the class that declares it).
- Only the 'right-hand side' of events are vulnerable to this.
    - If MyStaticClass was an object reference (myObjRef) it would not have been kept alive by the event subscription. This would allow myObject to be collected after myObjRef and the event delegate are collected – but would result in G3 promotion).
- As K2, applying this principal to our thick-client designers (K2 Designer for Visual Studio and K2 Studio), halved their typical memory usage.

Best practices – Assembly generating types

- Types such as XmlSerializer, Regex etc. generate assemblies at runtime.
- .NET cannot unload assemblies at runtime (unless in a separate unloaded AppDomain).
- Assemblies consume quite a bit of memory (>2MB before any code is even executed).
- Making these static read-only members of a class will only generate assemblies once.

```
public class MyXmlClass
{
  public static readonly XmlSerializer Serializer =
        new XmlSerializer(typeof(MyXmlClass));
  public static MyXmlClass Deserialize(XmlReader reader)
  {
    // Good memory implications.
    return (MyXmlClass)Serializer.Deserialize(reader);
  }

  public void Serialize(XmlWriter writer)
  {
    // Will leak memory rapidly.
    var serializer = new XmlSerializer(typeof(MyXmlClass));
    serializer.Serialize(this, writer);
  }
}
```

- K2 Workspace reports used to frequently exceed IIS memory limitations. We improved memory usage hugely by applying this to the XsltTransformation type.

Best practices – Roots (and static properties)

- The GC uses several ways to determine if an object can be seen by ‘something’.
- Certain objects are considered ‘roots’; these are objects that could be seen by executing code. The GC uses these roots to determine what to clean up next.
- Static properties are always roots.
- K2 now runs process definitions in a sandboxed environment; so when a process definition is no longer being used (i.e. there are no instances running against the definition) it will be cleaned up (including any static properties it may have). Nonetheless be very careful when using static properties.

OTHER READING

K2 Resources

- http://help.k2.com/en/KB000589.aspx (K2 Performance and Scalability white paper – based on 807 KB450)
- http://help.k2.com
- http://www.k2underground.com

Memory Leak Detection

- http://msdn.microsoft.com/en-us/magazine/cc163491.aspx
- http://www.codeproject.com/KB/dotnet/Memory_Leak_Detection.aspx
- http://madgeek.com/Articles/Leaks/Leaks.en.html

MSDN

- http://msdn.microsoft.com/en-us/library/ms998530 (outdated, but many very useful tips) http://msdn.microsoft.com/en-us/library/aa970850(v=VS.90).aspx (example of new approaches to memory in .NET)