We batched up our COM requests and return a single stream of results, but the performance is still slow

devblogs.microsoft.com/oldnewthing/20160212-00

February 12, 2016

Raymond Chen

A customer had a performance problem with an interface that connected to a service: The design was too chatty. ("Chatty" is a technical term that means "Too much time is spent communicating back and forth, often at the expense of actual work.") The original design went like this:

```
// All error checking deleted for expository purposes.
void
GetIdsAndNamesOfEverythingInContainer(IContainer* container)
{
  CComPtr<IIEnumItem> enumerator;
  container->GetEnumerator(&enumerator);
  for (CComPtr<IItem> item;
       enumerator->Next(&item) == S_OK;
       item.Release()) {
    UINT id;
    item->GetId(&id);
    CComHeapPtr<wchar_t> name;
    item->GetName(&name);
    AddToUI(id, name);
  }
}
```

If there are 10,000 items, then the number of trips to the server comes out to

Line	Calls to server
`GetEnumerator`	1
`Next`	10,001
`GetId`	10,000
`GetName`	10,000
`item->Release`	10,000

`enumerator->Release`	1
Total	40,003

If every call to the server takes one millisecond to complete, that's forty seconds spent collecting the IDs and names of all the items. (In this particular case, the server was local, but the high level of chattiness made the problem noticeable even for a local server.)

They reduced the chattiness by adding a special `GetIdsAndNamesOfAllChildren` method to perform a bulk operation. On the server side, it collects all the information and serializes it into a stream, then it returns the stream to the client. The client can then deserialize the data from the stream. Something like this:

```
// All error checking deleted for expository purposes.
void
GetIdsAndNamesOfEverythingInContainer(IContainer* container)
{
  // Issue the bulk request for the IDs and names of all children.
  CComPtr<IIStream> stream;
  container->GetIdsAndNamesOfAllChildren(&stream);

  // Now parse out the results.

  // First thing in the stream is the number of items.
  ULONG bytesRead;
  UINT count;
  stream->Read(&count, sizeof(count), &bytesRead);

  // For each item, read the ID and name
  for (UINT i = 0; i < count; i++) {
    UINT id;
    stream->Read(&id, sizeof(id), &bytesRead);

    // The string is preceded by a character count.
    UINT length;
    stream->Read(&length, sizeof(length), &bytesRead);

    // Yes, there is an integer overflow here - like I said,
    // I removed error checking for expository purposes.
    UINT byteLength = (length + 1) * sizeof(wchar_t);
    CComHeapPtr<wchar_t> name(CoTaskMemAlloc(byteLength));
    ZeroMemory(name, byteLength);

    stream->Read(name.m_pData, byteLength, &bytesRead);

    AddToUI(id, name);
  }
}
```

But the operation was still slow.

Let's study how chatty this new design is:

Line	Calls to server
`GetIdsAndNames`	1
`IStream::Read` for count	1
`IStream::Read` for Id	10,000
`IStream::Read` for length	10,000

`IStream::Read` for string	10,000
`IStream::Release`	1
Total	30,003

It's still ridiculously chatty! (We traded calls to `Next` and `Release` for a single call to get the length in the inner loop, but the other calls are still there.)

The problem is that the stream is marshaled by reference. When the COM marshaler returns the stream, it returns a proxy that talks back to the stream on the server. COM doesn't have any special knowledge about how you're using the stream. If you issue a read on the stream, COM marshal the read call back to the original object so that it can perform the read, which may consist of generating data on the fly or calling out to other objects, and it will certainly update the position of the stream pointer. Or maybe the stream on the server is constantly changing, so the read needs to retrieve the current data in the stream, even if it changed after the call to `GetIdsAndNamesOfAllChildren` returned.

The solution here is to bring domain-specific knowledge to the table. We know that the stream being returned is immutable, and it's not being shared with anybody. Indeed, once the server generates the output stream and returns it to the client, the server throws the stream away! What we want to happen is to transfer the *contents* of the stream to the client, so that the client gets a clone of the stream. Once that's done, all the stream operations on the client can be performed without having to talk back to the server.

One solution is to make the stream *marshal by value* by implementing *IMarshal* and providing a custom marshaler. Marshaling by value is common for immutable objects, because you can just transfer the object's state to the client, and then you're done. The client can talk to its local copy of the object instead of having to go back to the server all the time.

Another solution is to make the marshaling by value explicit by returning a block of memory rather than a COM object. Annotating your interface to indicate this is rather tricky, using the wacky `size_is(, ...)` syntax, where there is nothing between the open parenthesis and the comma. Once the client gets the raw buffer, it can parse the buffer directly, or it can create an `IStream` wrapper around it. (You might choose to create an `IStream` wrapper so that you minimize change to code that you've already spent time writing and debugging.)

Here's the revised table once we marshal the buffer by value, so that all of the `IStream` operations can be performed on the client side.

Line	Calls to server
`GetIdsAndNames`	1
`IStream::Read` for count	0
`IStream::Read` for Id	0
`IStream::Read` for length	0
`IStream::Read` for string	0
`IStream::Release`	0
Total	1

Bonus reading: Larry Osterman explains some of the nuances of `size_is` and `length_is`.

Update: Math is hard. Let's go shopping.

Raymond Chen

Follow