Protecting a broker from a failing delegate

devblogs.microsoft.com/oldnewthing/20230421-00

April 21, 2023

Raymond Chen

Last time, we saw how we can protect a broker from a failed event handler. But what if the thing you need to protect against is a delegate?

You can just replicate the event logic for detecting broken delegates, which we saw at the start of this mini-series.

```
// C++/CX
public delegate String^ SomeClassConversionHandler(Object^ o);

public ref class SomeClass
{
public:
  SomeClass(SomeClassConversionHandler^ converter) :
    m_converter(converter) {}

private:
  SomeClassConversionHandler^ m_converter;
};

// C++/WinRT
// .idl
namespace Contoso
{
  delegate String SomeClassConversionHandler(Object o);

  runtimeclass SomeClass
  {
    SomeClass(SomeClassConversionHandler converter);
  }
}

// .cpp
struct SomeClass : SomeClassT<SomeClass>
{
  SomeClass(Contoso::SomeClassConversionHandler const& converter)
    : m_converter(converter) {}

  Contoso::SomeClassConversionHandler m_converter;
};
```

Suppose the `SomeClass` object does some work, but occasionally needs help converting an object to a string, and that assistance comes from a `converter` provided at construction. Suppose the original code looks like this:

```
void SomeClass::DoWork()
{
  if (need_conversion) {
    auto converted = m_converter(o);
    /* ... more code ... */
  }
}
```

And we need to protect the call to `m_converter` from a failed delegate.

Well, first of all, we have to decide what we want to happen if the delegate fails.

One possibility is that we want to treat any kind of failure to mean that the conversion to a string produces an empty string.

```
// C++/CX
void SomeClass::DoWork()
{
  if (need_conversion) {
    String^ converted;
    try {
      converted = m_converter(o);
    } catch (...) {
      // Treat all conversion failures as
      // converting to empty string.
      converted = L"";
    }
    /* ... more code ... */
  }
}

// C++/WinRT
void SomeClass::DoWork()
{
  if (need_conversion) {
    winrt::hstring converted;
    try {
      converted = m_converter(o);
    } catch (...) {
      // Treat all conversion failures as
      // converting to empty string.
      converted = L"";
    }
    /* ... more code ... */
  }
}
```

Now, this does mean that if the failure is due to a disconnected delegate, every conversion attempt will raise a new `DisconnectedException` that is caught and ignored. This is rather inefficient, because disconnection is a permanent state, and it clutters the error logs, so you may want to detect disconnected converters as a special case.

```
bool IsDisconnectedHResult(int hr)
{
  return hr == HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE) ||
         hr == RPC_E_DISCONNECTED ||
         hr == JSCRIPT_E_CANTEXECUTE;
}

// C++/CX
void SomeClass::DoWork()
{
  if (need_conversion) {
    String^ converted;
    if (m_converted == nullptr) {
      // If no converter, then just use empty string.
      converted = L"";
    } else {
      try {
        converted = m_converter(o);
      } catch (Exception^ ex) {
        if (IsDisconnectedHResult(ex->HResult)) {
          // Don't use this disconnected converter any more.
          m_converter = nullptr;
        }
        // Treat all conversion failures as
        // converting to empty string.
        converted = L"";
      }
    }
    /* ... more code ... */
  }
}

// C++/WinRT
void SomeClass::DoWork()
{
  if (need_conversion) {
    winrt::hstring converted;
    if (m_converted == nullptr) {
      // If no converter, then just use empty string.
      converted = L"";
    } else {
      try {
        converted = m_converter(o);
      } catch (...) {
        if (IsDisconnectedHResult(winrt::to_hresult())) {
          // Don't use this disconnected converter any more.
          m_converter = nullptr;
        }
        // Treat all conversion failures as
        // converting to empty string.
        converted = L"";
      }
    }
    /* ... more code ... */
  }
}
```

Here, we overload `nullptr` as a special converter value to mean "Don't even try converting. Just go straight to an empty string."

If you already have assigned a special meaning to `nullptr` (say, because that means "Perform a default conversion"), then you have some other options.

You could add another member variable to remember whether the converter is any good.

```
// C++/CX
public ref class SomeClass
{
public:
  SomeClass(SomeClassConversionHandler^ converter) :
    m_converter(converter) {}

private:
  SomeClassConversionHandler^ m_converter;
  bool m_isConverterDisconnected = false;
};

void SomeClass::DoWork()
{
  if (need_conversion) {
    String^ converted;
    if (m_isConverterDisconnected) {
      // Disconnected converter is treated as converting
      // to empty string.
      converted = L"";
    } else if (m_converter == nullptr) {
      converted = DefaultConversion(o);
    } else {
      try {
        converted = m_converter(o);
      } catch (Exception^ ex) {
        if (IsDisconnectedHResult(ex->HResult)) {
          // Don't use this disconnected converter any more.
          m_isConverterDisconnected = true;
        } else {
          // Treat all conversion failures as
          // converting to empty string.
          converted = L"";
        }
      }
    }
    /* ... more code ... */
  }
}

// C++/WinRT
struct SomeClass : SomeClassT<SomeClass>
{
  SomeClass(Contoso::SomeClassConversionHandler const& converter)
    : m_converter(converter) {}

  Contoso::SomeClassConversionHandler m_converter;
  bool m_isConverterDisconnected = false;
};

void SomeClass::DoWork()
{
  if (need_conversion) {
    winrt::hstring converted;
    if (m_isConverterDisconnected) {
      // Disconnected converter is treated as converting
      // to empty string.
      converted = L"";
    } else if (m_converter == nullptr) {
      converted = DefaultConversion(o);
    } else {
      try {
        converted = m_converter(o);
      } catch (...) {
        if (IsDisconnectedHResult(winrt::to_hresult())) {
          // Don't use this disconnected converter any more.
          m_isConverterDisconnected = true;
        } else {
          // Treat all conversion failures as
          // converting to empty string.
          converted = L"";
        }
      }
    }
    /* ... more code ... */
  }
}
```

Or you could replace the disconnected delegate with a working one that performs the “What to do if the delegate is disconnected” action.

```
// C++/CX
void SomeClass::DoWork()
{
  if (need_conversion) {
    String^ converted;
    if (m_converter == nullptr) {
      converted = DefaultConversion(o);
    } else {
      try {
        converted = m_converter(o);
      } catch (Exception^ ex) {
        if (IsDisconnectedHResult(ex->HResult)) {
          // Replace this disconnected delegate with a dummy one.
          m_converter = ref new SomeClassConversionHandler(
            [](Object^) -> String^ { return L""; });
        }
        // Treat all conversion failures as
        // converting to empty string.
        converted = L"";
      }
    }
    /* ... more code ... */
  }
}

// C++/WinRT
void SomeClass::DoWork()
{
  if (need_conversion) {
    winrt::hstring converted;
    if (m_converter == nullptr) {
      converted = DefaultConversion(o);
    } else {
      try {
        converted = m_converter(o);
      } catch (...) {
        if (IsDisconnectedHResult(winrt::to_hresult())) {
          // Replace this disconnected delegate with a dummy one.
          m_converter = [](auto&&) { return winrt::hstring(); };
        }
        // Treat all conversion failures as
        // converting to empty string.
        converted = L"";
      }
    }
    /* ... more code ... */
  }
}
```

Now, maybe you want to deal with disconnected delegates differently from actively broken ones. For example, the delegate might throw an `E_INVALIDARG` if the object is not convertible at all, and that’s different from “converts to nothing”.

```
// C++/CX
void SomeClass::DoWork()
{
  if (need_conversion) {
    String^ converted;
    try {
      if (m_converter == nullptr) {
        converted = DefaultConversion(o);
      } else {
        converted = m_converter(o);
      }
    } catch (Exception^ ex) {
      if (IsDisconnectedHResult(ex->HResult)) {
        // Don't use this disconnected converter any more.
        // Use default conversions from now on.
        m_converter = nullptr;
        converted = DefaultConversion(o);
      } else {
        throw;
      }
    }
    /* ... more code ... */
  }
}

// C++/WinRT
void SomeClass::DoWork()
{
  if (need_conversion) {
    winrt::hstring converted;
    try {
      if (m_converter == nullptr) {
        converted = DefaultConversion(o);
      } else {
        converted = m_converter(o);
      }
    } catch (...) {
      if (IsDisconnectedHResult(winrt::to_hresult())) {
        // Don't use this disconnected converter any more.
        // Use default conversions from now on.
        m_converter = nullptr;
        converted = DefaultConversion(o);
      } else {
        throw;
      }
    }
    /* ... more code ... */
  }
}
```

Here, we rethrow any exception from the converter that isn't a disconnection. The exception then propagates back to the caller, to tell it that the conversion failed. The caller can then take whatever remedial action it deems appropriate.

Dealing with a broken delegate is more work than dealing with a broken event handler because you have to reimplement the disconnection logic yourself. One way to sidestep this problem is to to leverage all the work that went into events: Just use an event!

```
// C++/CX
public ref class SomeClassConversionRequestedEventArgs
{
    property Object^ Value { Object^ get(); };
    property String^ Converted
    { String^ get(); void set(String^); };
};

public ref class SomeClass
{
public:
  SomeClass() {}

  event TypedEventHandler<SomeClass^, SomeClassConversionRequestedEventArgs^>;^
    ConversionRequested;
};

void SomeClass::DoWork()
{
  if (need_conversion) {
    auto args = ref new SomeClassConversionRequestedEventArgs();
    args->Value = o;
    ConversionRequested(this, args);
    auto converted = args->Converted;
    /* ... more code ... */
  }
}


// C++/WinRT
// .idl
namespace Contoso
{
  runtimeclass SomeClassConversionRequestedEventArgs
  {
    Object Value{ get };
    String Converted;
  }

  runtimeclass SomeClass
  {
    SomeClass();
    event TypedEventHandler<SomeClass, SomeClassConversionRequestedEventArgs>
      ConversionRequested;
  }
}

// .cpp
struct SomeClass : SomeClassT<SomeClass>
{
  SomeClass() {}
  /* save some typing */
  using ConversionHandler = winrt::TypedEventHandler<
      Contoso::SomeClass, Contoso::SomeClassConversionRequestedEventArgs>;

  auto ConversionRequested(ConversionHandler const& handler) {
    return m_conversionRequestedEvent.add(handler);
  }
  void ConversionRequested(winrt::event_token const& token) {
    return m_conversionRequestedEvent.remove(token);
  }

  winrt::event<
    winrt::TypedEventHandler<
      Contoso::SomeClass, Contoso::SomeClassConversionRequestedEventArgs>>
    m_conversionRequestedEvent;
};

void SomeClass::DoWork()
{
  if (need_conversion) {
    Contoso::SomeClassConversionRequestedEventArgs args;
    args.Value(o);
    m_conversionRequestedEvent(*this, args);
    auto converted = args.Converted();
    /* ... more code ... */
  }
}
```

The caller registers an event handler for the `ConversionRequested` event, and now the event infrastructure now does the work of dealing with the disconnection exception. You can then apply the other techniques for protecting a broker from a failed event handler if you want to customize how errors are handled.