Protecting a broker from a failing event handler

devblogs.microsoft.com/oldnewthing/20230420-00

April 20, 2023

Raymond Chen

Last time, we identified the source of a crash as an unhandled exception from an out-of-process event handler. What sort of defense is appropriate against misbehaving event handlers?

First, the easy case: If you event source accepts handlers only from within the same process, then you don't need to have any defense at all. If an event handler fails, then you can just propagate the failure. After all, if that event handler crashed, then you would have crashed, since you're all running in the same process and therefore at the same security level. Let the crash get caught by your program's reliability infrastructure so it can be investigated like any other crash.

Okay, so that leaves the out-of-process handlers.

Does this broker service a single client, or can it have multiple clients? For example, if two UWP apps want to perform the operation mediated by the broker, does each app get its own broker? Or is a single broker shared by all the apps?

If each app gets its own broker, then you can let the crashed client crash the broker. After all, when the client crashes, the system will (eventually) shut down the broker anyway, since there is nothing left for the broker to do. Crashing immediately just accelerates the cleanup.

On the other hand, if a single broker can service multiple clients, then it makes sense to protect itself from misbehaving clients, because one failed client shouldn't affect other clients.

We can protect ourselves from failed clients by using a custom wrapper that deals with client errors. I'll start with the C++/CX version (since that's what our customer was using), and then port to C++/WinRT later. All of our wrappers follow this pattern:

```
template<typename Handler>
Handler^ WrapHandler(Handler^ handler)
{
    return ref new Handler([=](auto... args)
    {
        try {
            handler(args...);
        } catch (Platform::Exception^ ex) {
            /* stuff will go here */
        }
    }
}

public ref class SomeClass
{
    /* save some typing */
    using Windows::Foundation::EventRegistrationToken;
    using ChangedHandler = EventHandler<Platform::Object^>;

    event ChangedHandler^ Changed {
        EventRegistrationToken add(ChangedHandler^ handler) {
            return ChangedInternal += WrapHandler(handler);
        }
        void remove(EventRegistrationToken token) {
            return ChangedInternal -= token;
        }
    };

private:
    event ChangedHandler^ ChangedInternal;

    void NotifyChanged() { ChangedInternal(this, nullptr); }
};
```

We create a public event called `Changed`, but this is just a façade for the private one `ChangedInternal`. When a client adds a handler to the `Changed` event, we wrap it and add the wrapped handler to the private `ChangedInternal` event. Not that we let the original `ChangedInternal` manage the event token.

If an error occurs in the client's handler (which is projected into C++/CX as an exception), we catch the exception, and now the excitement begins. What we do with the exception depends on how we want to treat client errors.

One policy might be to ignore all client errors, but leave the handler registered for subsequent events. In that case, we want to swallow all exceptions and tell the eventing infrastructure, "Move along now, nothing to see here."

```
template<typename Handler>
Handler^ WrapHandler(Handler^ handler)
{
    return ref new Handler([=](auto... args)
    {
        try {
            handler(args...);
        } catch (Platform::Exception^ ex) {
            /* ignore all errors */
        }
    }
}
```

This is probably not a great idea, because it means that if the handler process crashes, we ignore the `DisconnectedException`, and the handler never unregisters. (The code that would unregister the handle has already crashed, so the handler has been leaked.) If you get a rash of clients connecting and crashing (say, due to some sort of crash loop), the broker will slowly leak data and eventually exhaust memory, probably taking out not only itself, but other processes which receive "out of memory" errors. the

So really, we should allow the "handler is permanently dead" exceptions to escape, so that the event infrastructure will clean it up.

```
template<typename Handler>
Handler^ WrapHandler(Handler^ handler)
{
    return ref new Handler([=](auto... args)
    {
        try {
            handler(args...);
        } catch (Platform::Exception^ ex) {
            // Rethrow expired handler exceptions.
            HRESULT hr = hr->HResult;
            if (hr == RPC_E_DISCONNECTED ||
                hr == HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE) ||
                hr == JSCRIPT_E_CANTEXECUTE) throw;
            // Ignore all other errors.
        }
    }
}
```

Yet another policy is to force-unregister a handler at the first sign of trouble. Any exception from a handler causes it to be removed.

```
template<typename Handler>
Handler^ WrapHandler(Handler^ handler)
{
    return ref new Handler([=](auto... args)
    {
        try {
            handler(args...);
        } catch (Platform::Exception^ ex) {
            // Force this handler to expire on any exception.
            throw ref new Platform::DisconnectedException();
        }
    }
}
```

We can encapsulate this into a policy object, and add calls to `RoTransformError` to aid in post-mortem debugging.

```
struct IgnoreAllPolicy
{
    static void OnException(HRESULT) { }
};

struct IgnoreErrorsPolicy
{
    static void OnException(HRESULT hr) {
        if (hr == RPC_E_DISCONNECTED ||
            hr == HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE) ||
            hr == JSCRIPT_E_CANTEXECUTE) throw;
    }
};

struct DisconnectOnErrorPolicy
{
    static void OnException(HRESULT hr) {
        RoTransformErrorW(hr, RPC_E_DISCONNECTED,
            0, L"Disconnecting on all errors");
        throw ref new Platform::DisconnectedException();
    }
};

template<typename Policy, typename Handler>
Handler^ WrapHandler(Handler^ handler)
{
    return ref new Handler([=](auto... args)
    {
        try {
            handler(args...);
        } catch (Platform::Exception^ ex) {
            Policy::OnException(ex->HResult);
            RoTransformErrorW(ex->HResult, S_OK,
                0, L"Ignoring delegate error");
        }
    }
}
```

The `RoTransformErrorW` function tells the debugging infrastructure "There was some sort of error, but I'm changing it to another kind of error (or a success)." We report a transformation to `S_OK` to say, "It's not an error after all. If you see a crash or unhandled exception soon afterward, don't blame me!" And we report a transformation to `RPC_E_DISCONNECTED` to let debuggers stitch together exception history. If somebody is looking for the history of an `RPC_E_DISCONNECTED` exception, and they trace it back to our function, the transformation tells them to consider this exception and the earlier exception to be part of the same exception history.

Here's a sample usage:

```
public ref class SomeClass
{
    ...

    EventRegistrationToken add(ChangedHandler^ handler) {
        return ChangedInternal +=
            WrapHandler<DisconnectOnErrorPolicy>(handler);
    }

    ...
};
```

Okay, now on to C++/WinRT. It's a fairly straightforward port.

```
struct IgnoreAll
{
    static void OnException(winrt::hresult) { }
};

struct IgnoreErrors
{
    static void OnException(winrt::hresult hr) {
        if (hr == RPC_E_DISCONNECTED ||
            hr == HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE) ||
            hr == JSCRIPT_E_CANTEXECUTE) throw;
    }
};

struct DisconnectOnError
{
    static void OnException(winrt::hresult hr) {
        RoTransformErrorW(hr, RPC_E_DISCONNECTED,
            0, L"Disconnecting on all errors");
        throw winrt::hresult_error(RPC_E_DISCONNECTED);
    }
};

template<typename Policy, typename Handler>
Handler WrapHandler(Handler&& handler)
{
    return [handler = std::forward<Handler>(handler)]
           (auto&&... args)
    {
        try {
            handler(std::forward<decltype(args)>(args)...);
        } catch (...) {
            auto hr = winrt::to_hresult();
            Policy::OnException(hr);
            RoTransformErrorW(hr, S_OK,
                0, L"Ignoring delegate error");
        }
    }
}

// sample usage

winrt::event_token Changed(ChangedHandler const& handler)
{
    return m_changedInternal(WrapHandler<DisconnectOnErrorPolicy>(handler));
}

winrt::event_token Changed(ChangedHandler const& handler)
{
    return m_changedInternal(WrapHandler<IgnoreErrorsPolicy>(handler));
}
```

Now, the `winrt::event` object already ignores all non-expiring exceptions, so if you are wrapping the handler for something that you know is a `winrt::event`, then the only custom wrapper you need is for disconnecting on any error.

```
template<typename Handler>
Handler DisconnectOnError(Handler const& handler)
{
    return [=](auto&&... args)
    {
        try {
            handler(std::forward<decltype(args)>(args)...);
        } catch (...) {
            RoTransformErrorW(winrt::to_hresult(), RPC_E_DISCONNECTED,
                0, L"Disconnecting on all errors");
            throw winrt::hresult_error(RPC_E_DISCONNECTED);
        }
    };
}

// sample usage

winrt::event_token Changed(ChangedHandler const& handler)
{
    return m_changedInternal(DisconnectOnError(handler));
}
```

Bonus chatter: You might realize that `WrapHandler` returns a lambda, which is just a class, so maybe we can make `WrapHandler` itself a class:

```
template<typename Policy, typename Handler>
struct WrapHandler
{
    WrapHandler(Handler&& handler)
    : m_handler(std::move(handler)) {}

    WrapHandler(Handler const& handler)
    : m_handler(handler) {}

    template<typename... Args>
    auto operator()(Args&&... args)
    {
        try {
            handler(std::forward<Args>(args)...);
        } catch (...) {
            auto hr = winrt::to_hresult();
            Policy::OnException(hr);
            RoTransformErrorW(hr, S_OK,
                0, L"Ignoring delegate error");
        }
    }
private:
    Handler m_handler;
};

template<typename Handler>
using DisconnectOnHandlerErrors = WrapHandler<DisconnectOnErrorPolicy, Handler>;
template<typename Handler>
using IgnoreHandlerErrors = WrapHandler<IgnoreErrorsPolicy, Handler>;
```

The `WrapHandler` class works okay, but the type aliases don't work because you cannot partially specialize an alias template.