Real-time Trade-offs

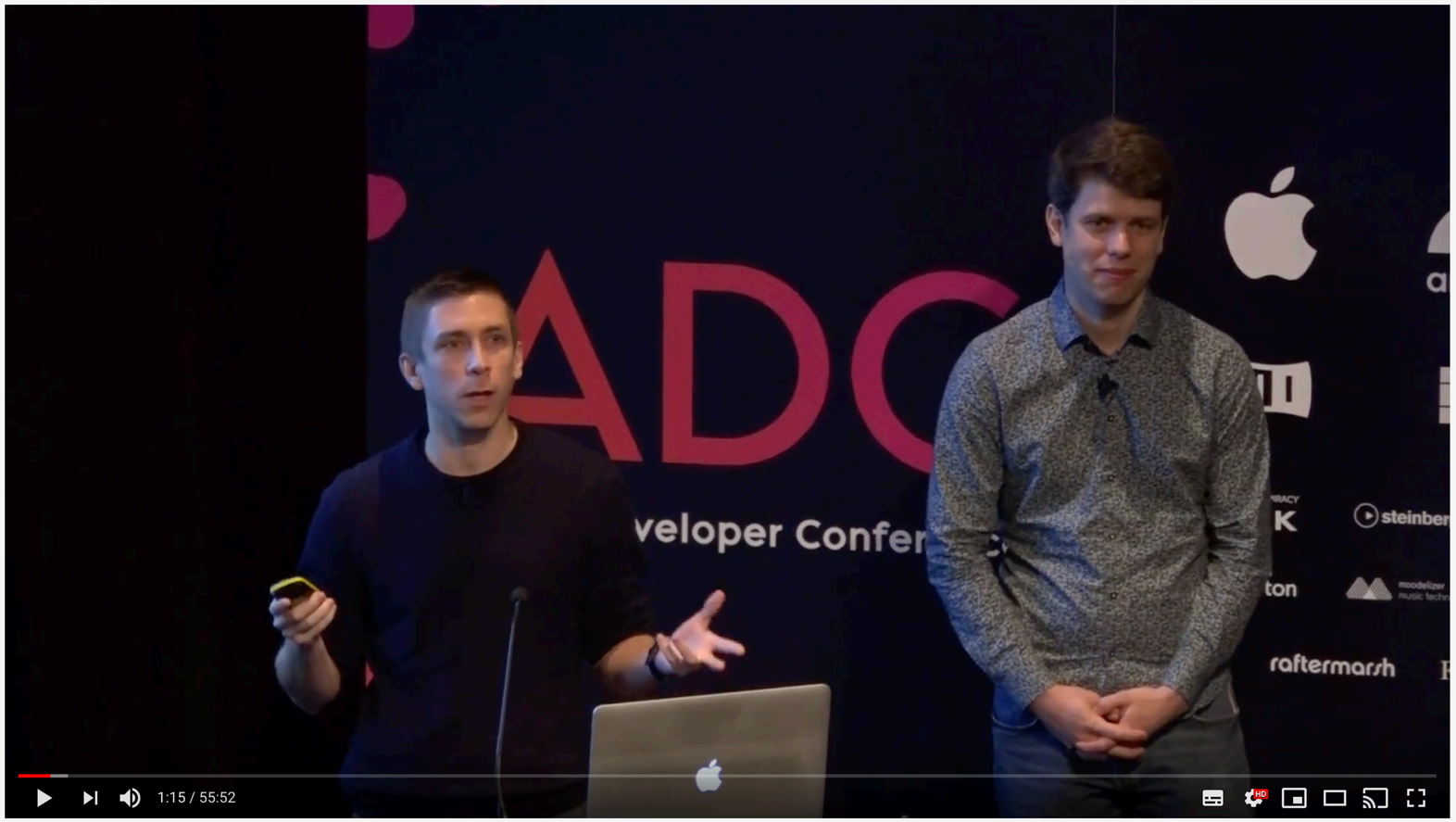
David Rowland

@drowaudio

Spoilers

- There won’t be a “right” answer

- Real-time 101 - David Rowland & Fabien Renn-Giles
  ADC 2019



1. Background

APVTS Updates & Thread/Realtime Safety

RustyPine Jan 3

I've got a few questions on this topic so I numbered them. Very curious how others have dealt with these issues.

1. In my various freelance work I've inherited projects more than once where `AudioProcessorValueTreeState::Listener::parameterChanged` is overridden in a way where the previous dev(s) made calls to gui related code (e.g. `Label::setText`). This isn't safe since `parameterChanged` is often called on the audio thread for automation purposes (which can also be tested with PluginVal). Perhaps we can make it more obvious that it isn't safe? Maybe put something in the docs to dissuade this usage?

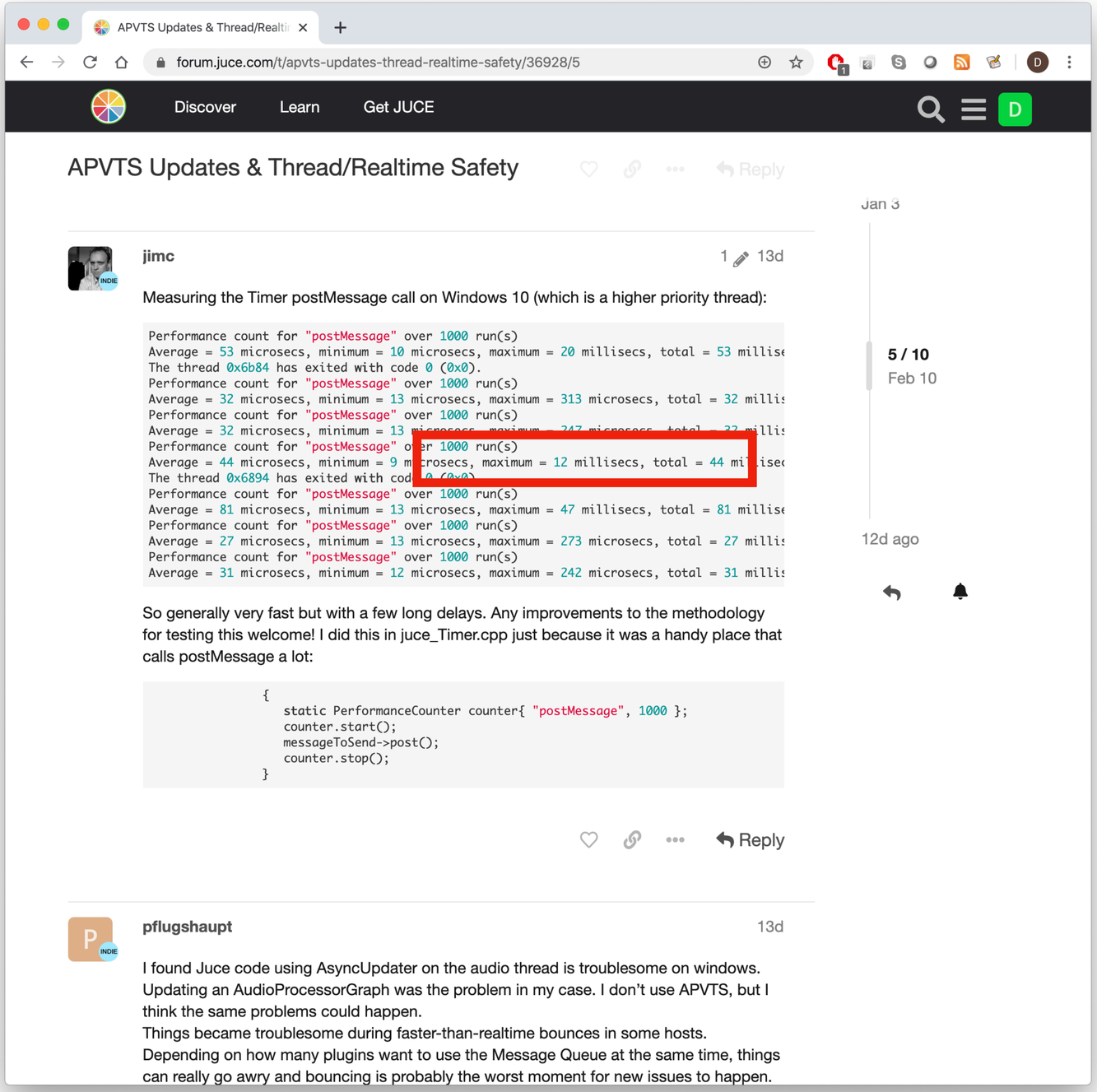
2. To safely connect gui to the parameter system an obvious solution is to use one of the attachments provided. However when I look into the code to see how it's implemented it's using an `AsyncUpdater` which allocates when triggered and is thus not realtime safe:

```
void AttachedControlBase::parameterChanged (const String&, float newValue) override
{
    lastValue = newValue;

    if (MessageManager::getInstance()->isThisTheMessageThread())
    {
        cancelPendingUpdate();
        setValue (newValue);
    }
    else
    {
        triggerAsyncUpdate();
    }
}
```

Is there something I'm missing here?

3. To me, the next obvious way to connect gui stuff to the parameters without using the attachments is to listen to the APVTS's `ValueTree`. Looking under the hood this uses atomic flags and a main thread timer to poll for updates, so it looks like the thread and realtime safe solution I want:

```
void AudioProcessorValueTreeState::timerCallback()
{
    auto anythingUpdated = flushParameterValuesToValueTree();

    startTimer (anythingUpdated ? 1000 / 50
                                : jlimit (50, 500, getTimerInterval() + 20));
```


APVTS Updates & Thread/Realti
forum.juce.com/t/apvts-updates-thread-realtime-safety/36928/5
Discover
Learn
Get JUCE
APVTS Updates & Thread/Realtime Safety
Reply
Jan 3
jimc
1
13d
Measuring the Timer postMessage call on Windows 10 (which is a higher priority thread):
Performance count for "postMessage" over 1000 run(s)
Average = 53 microsecs, minimum = 10 microsecs, maximum = 20 millisecs, total = 53 millise
The thread 0x6b84 has exited with code 0 (0x0).
Performance count for "postMessage" over 1000 run(s)
Average = 32 microsecs, minimum = 13 microsecs, maximum = 313 microsecs, total = 32 millis
Performance count for "postMessage" over 1000 run(s)
Average = 32 microsecs, minimum = 13
Performance count for "postMessage" o
Average = 44 microsecs, minimum = 9
The thread 0x6894 has exited with cod
Performance count for "postMessage" over 1000 run(s)
Average = 81 microsecs, minimum = 13 microsecs, maximum = 47 millisecs, total = 81 millise
Performance count for "postMessage" over 1000 run(s)
Average = 27 microsecs, minimum = 13 microsecs, maximum = 273 microsecs, total = 27 millis
Performance count for "postMessage" over 1000 run(s)
Average = 31 microsecs, minimum = 12 microsecs, maximum = 242 microsecs, total = 31 millis
5 / 10
Feb 10
12d ago
So generally very fast but with a few long delays. Any improvements to the methodology for testing this welcome! I did this in juce_Timer.cpp just because it was a handy place that calls postMessage a lot:
{
static PerformanceCounter counter{ "postMessage", 1000 };
counter.start();
messageToSend->post();
counter.stop();
}
Reply
pflugshaupt
13d
I found Juce code using AsyncUpdater on the audio thread is troublesome on windows. Updating an AudioProcessorGraph was the problem in my case. I don't use APVTS, but I think the same problems could happen.
Things became troublesome during faster-than-realtime bounces in some hosts.
Depending on how many plugins want to use the Message Queue at the same time, things can really go awry and bouncing is probably the worst moment for new issues to happen.

How long is an audio buffer?

~2ms

PostMessageA function

12/05/2018 • 2 minutes to read

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message.

To post a message in the message queue associated with a thread, use the PostThreadMessage function.

Syntax

C++ Copy

```
BOOL PostMessageA(
  HWND   hWnd,
  UINT   Msg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

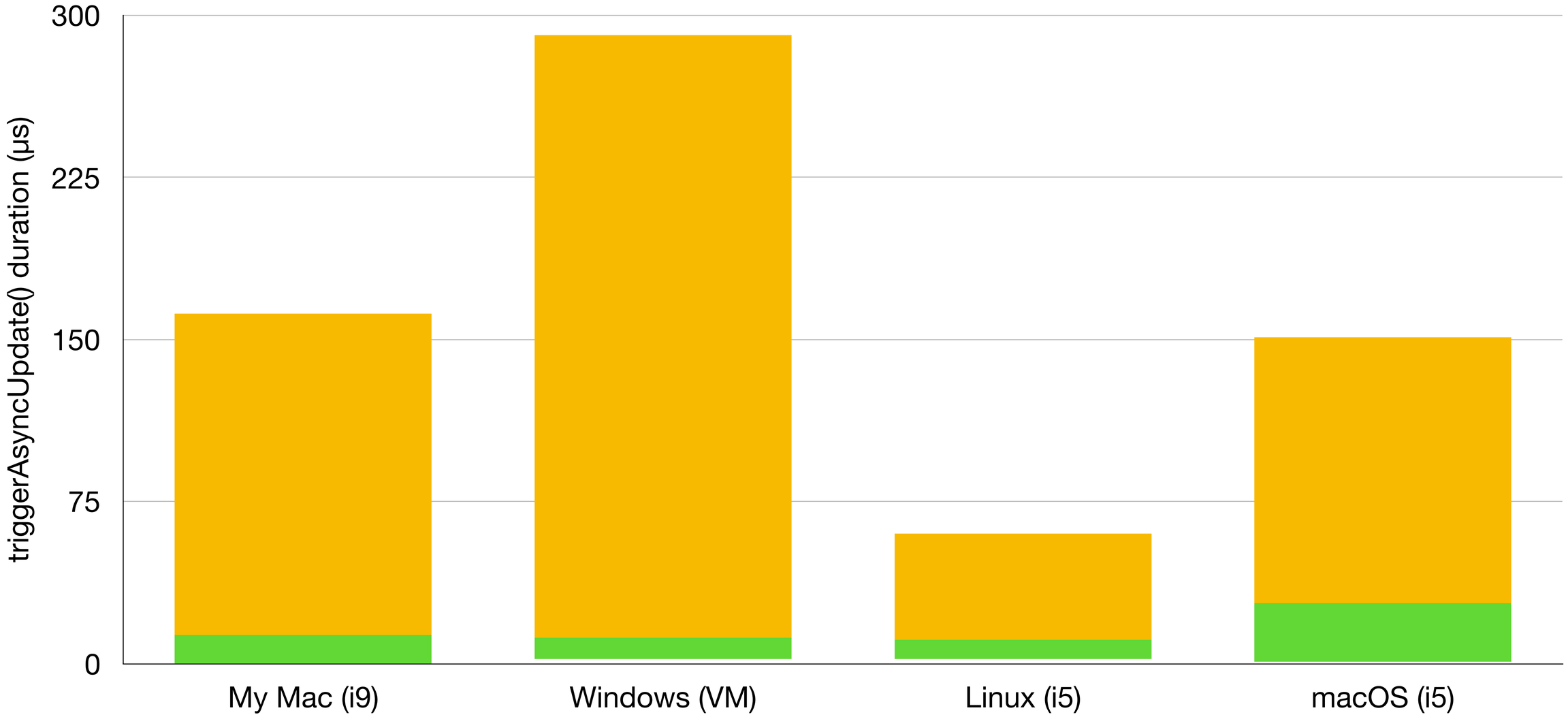
`hWnd`

Type: **HWND**

A handle to the window whose window procedure is to receive the message. The following values have special meanings.

Value	Meaning
HWND_BROADCAST ((HWND)0xffff)	The message is posted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows. The message is not posted to child windows.
NULL	The function behaves like a call to

Cost of `triggerAsyncUpdate()`



In Summary

- Don’t block!
  - **System calls**
  - **Waiting to acquire a lock** of any kind

2. What do we do?

Understand the problem

- When using **`juce::AudioProcessorValueTreeState`**, **`AudioProcessorParameter::setValue`** can get called from **`processBlock`** on the real-time thread

- Calls **`parameterValueChanged`** for any listeners

- **`AttachedControlBase::parameterChanged`** calls **`triggerAsyncUpdate`**

```
void AttachedControlBase::parameterChanged (const String&, float newValue) override
{
    lastValue = newValue;

    if (MessageManager::getInstance()->isThisTheMessageThread())
    {
        cancelPendingUpdate();
        setValue (newValue);
    }
    else
    {
        triggerAsyncUpdate();
    }
}
```

What can we do?

1. Not use anything that calls **`PostMessage`** from a real-time thread
2. Not use **`juce::AsyncUpdater`** from a real-time thread
3. Propose some solutions

3. Create a `RealTimeAsyncUpdater`

```
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class JUCE_API  AsyncUpdater
{
public:
    //==============================================================================
    /** Creates an AsyncUpdater object. */
    AsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~AsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread, BUT beware of calling
        it from a real-time (e.g. audio) thread, because it involves posting a message
        to the system queue, which means it may block (and in general will do on
        most OSes).
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //==============================================================================
    class AsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<AsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<AsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (AsyncUpdater)
};

} // namespace juce
```

```
    Basically, one or more calls to the triggerAsyncUpdate() will result in the
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class  RealTimeAsyncUpdater
{
public:
    //==============================================================================
    /** Creates a RealTimeAsyncUpdater object. */
    RealTimeAsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~RealTimeAsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread.
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //==============================================================================
    class RealTimeAsyncUpdateDispatcher;
    class RealTimeAsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RealTimeAsyncUpdater)
};

```

juce::AsyncUpdater

```
101 private:
102     //==============================================================================
103     class AsyncUpdaterMessage;
104     friend class ReferenceCountedObjectPtr<AsyncUpdaterMessage>;
105     ReferenceCountedObjectPtr<AsyncUpdaterMessage> activeMessage;
106
107     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (AsyncUpdater)
108 };
```

RealTimeAsyncUpdater

```
82 private:
83     //==============================================================================
84     class RealTimeAsyncUpdateDispatcher;
85     class RealTimeAsyncUpdaterMessage;
86     friend class ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage>;
87     ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage> activeMessage;
88
89     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RealTimeAsyncUpdater)
90 };
91
```

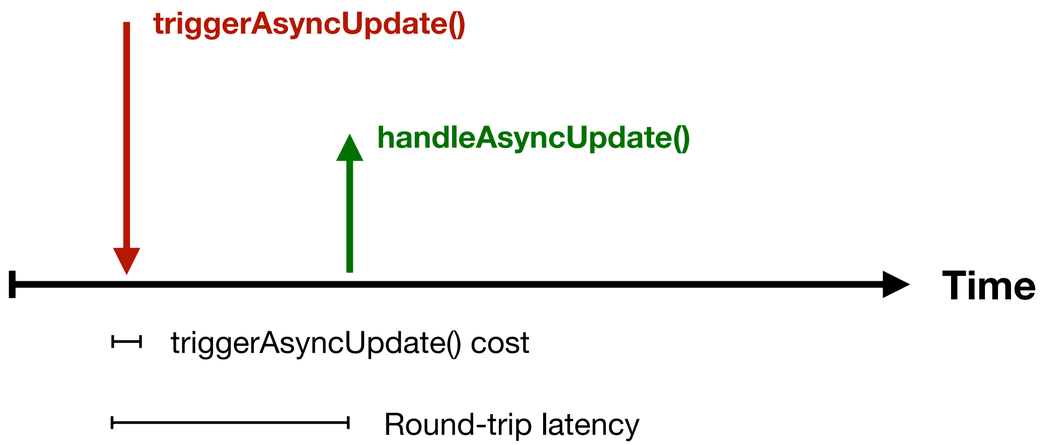
Write some tests

```
class RealTimeAsyncUpdaterTests : public juce::UnitTest
{
public:
    RealTimeAsyncUpdaterTests()
        : juce::UnitTest ("RealTimeAsyncUpdater", "Tracktion:Longer") {}

    //==============================================================================
    void runTest() override
    {
        if (MessageManager::getInstanceWithoutCreating() == nullptr)
            return;

        beginTest ("juce::AsyncUpdater");
        runAsyncUpdateTest<juce::AsyncUpdater>();
        beginTest ("RealTimeAsyncUpdater");
        runAsyncUpdateTest<RealTimeAsyncUpdater>();
    }

    template<typename UpdaterType>
    void runAsyncUpdateTest())
    {
       ...
    }

    template<typename UpdaterType>
    struct UpdaterTest : public UpdaterType
    {
       …
    };
};

static RealTimeAsyncUpdaterTests realTimeAsyncUpdaterTests;
```

```cpp
template<typename UpdaterType>
struct UpdaterTest : public UpdaterType
{
    UpdaterTest() = default;

    void sendUpdate()
    {
        hasDelivered = false;
        UpdaterType::triggerAsyncUpdate();
    }

    void handleAsyncUpdate() override
    {
        hasDelivered = true;
        event.signal();
        JUCE_ASSERT_MESSAGE_THREAD;
    }

    WaitableEvent event;
    std::atomic<bool> hasDelivered { false };
};
```

```
template<typename UpdaterType>
void runAsyncUpdateTest()
{
    UpdaterTest<UpdaterType> updater;
    PerformanceCounter pc ("RealTimeAsyncUpdaterCounter", 1000);
    std::atomic<bool> hasFinished { false };

    std::thread t ([&]
                   {
                       for (int i = 0; i < 10'000; ++i)
                       {
                           pc.start();
                           updater.sendUpdate();
                           updater.event.wait (-1);
                           pc.stop();

                           if (! updater.hasDelivered.load())
                               expect (false);
                       }

                       hasFinished = true;
                   });

    while (! hasFinished.load())
        MessageManager::getInstance()->runDispatchLoopUntil (5);

    t.join();
    expect (updater.hasDelivered.load());
}
```

Measuring message post -> callback latency

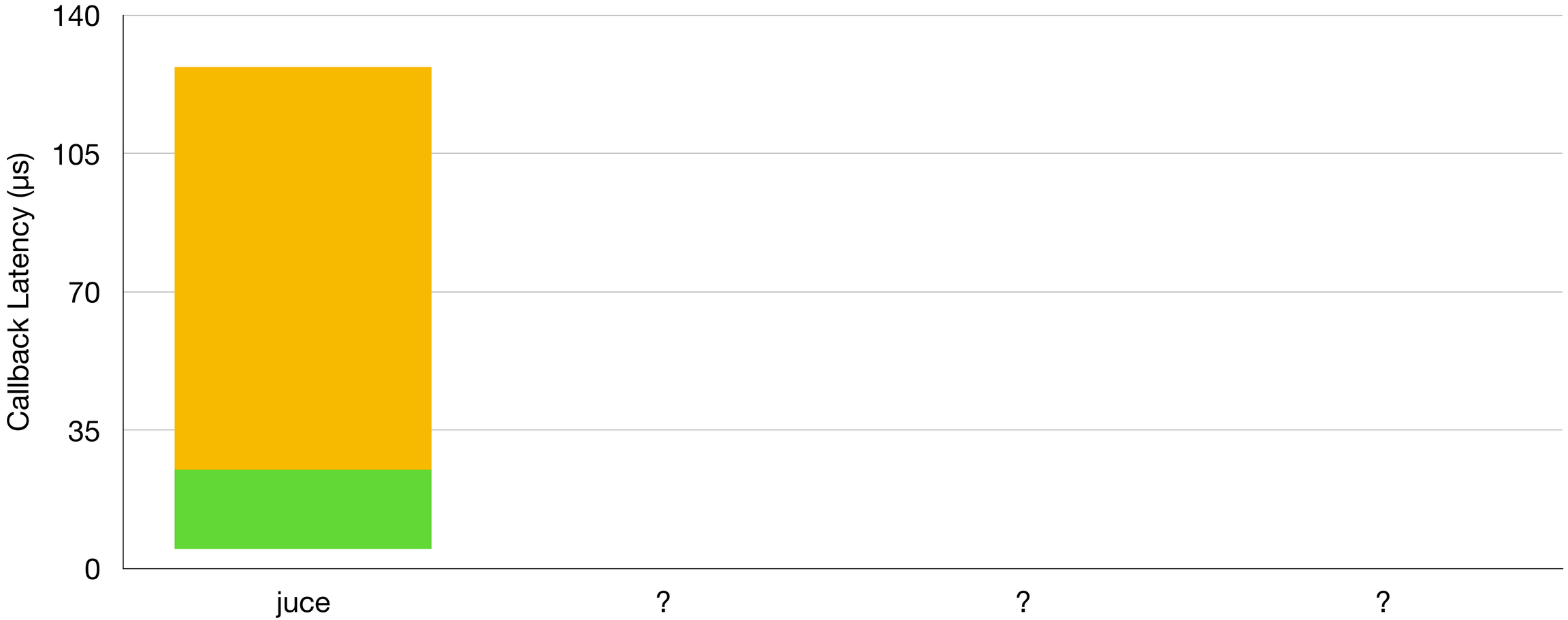

triggerAsyncUpdate()
handleAsyncUpdate()
Time
triggerAsyncUpdate() cost
Round-trip latency

juce::AsyncUpdater

```
Average = 21 microsecs, minimum = 6 microsecs, maximum = 52  microsecs, total = 21 millisecs
Average = 19 microsecs, minimum = 7 microsecs, maximum = 74  microsecs, total = 19 millisecs
Average = 20 microsecs, minimum = 5 microsecs, maximum = 99  microsecs, total = 20 millisecs
Average = 18 microsecs, minimum = 5 microsecs, maximum = 96  microsecs, total = 18 millisecs
Average = 21 microsecs, minimum = 6 microsecs, maximum = 97  microsecs, total = 21 millisecs
Average = 19 microsecs, minimum = 5 microsecs, maximum = 70  microsecs, total = 19 millisecs
Average = 20 microsecs, minimum = 5 microsecs, maximum = 102 microsecs, total = 20 millisecs
Average = 18 microsecs, minimum = 7 microsecs, maximum = 85  microsecs, total = 18 millisecs
Average = 18 microsecs, minimum = 7 microsecs, maximum = 65  microsecs, total = 18 millisecs
Average = 19 microsecs, minimum = 6 microsecs, maximum = 91  microsecs, total = 19 millisecs
```

Average = 20 microsecs, minimum = 5 microsecs, maximum = 102 microsecs

Round-trip Latency



Implement `RealTimeAsyncUpdater`

```cpp
//==============================================================================
AsyncUpdater::AsyncUpdater()
{
    activeMessage = *new AsyncUpdaterMessage (*this);
}

AsyncUpdater::~AsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
              || MessageManager::getInstanceWithoutCreating() == nullptr
              || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    if (activeMessage->shouldDeliver.compareAndSetBool (1, 0))
        if (! activeMessage->post())
            cancelPendingUpdate(); // if the message queue fails, this avoids
                                   // trapped waiting for the message to arriv
}

void AsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool AsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```cpp
//==============================================================================
RealTimeAsyncUpdater::RealTimeAsyncUpdater()
{
    activeMessage = *new RealTimeAsyncUpdaterMessage (*this);
}

RealTimeAsyncUpdater::~RealTimeAsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
              || MessageManager::getInstanceWithoutCreating() == nullptr
              || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    // Here we just set the atomic flag and wait for it to be serviced
    activeMessage->postUpdate();
}

void RealTimeAsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool RealTimeAsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```
class RealTimeAsyncUpdater::RealTimeAsyncUpdaterMessage  : public ReferenceCountedObject
{
public:
    RealTimeAsyncUpdaterMessage (RealTimeAsyncUpdater& au)
        : owner (au)
    {
        dispatcher->add (*this);
    }

    ~RealTimeAsyncUpdaterMessage()
    {
        dispatcher->remove (*this);
    }

    void postUpdate()
    {
        shouldDeliver.set (1);
    }

    void serviceMessage()
    {
        if (shouldDeliver.compareAndSetBool (0, 1))
            owner.handleAsyncUpdate();
    }

    RealTimeAsyncUpdater& owner;
    Atomic<int> shouldDeliver;
    SharedResourcePointer<RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher> dispatcher;

    JUCE_DECLARE_NON_COPYABLE (RealTimeAsyncUpdaterMessage)
};
```

Real-time safe

Trade-offs

- We now have a real-time safe way of posting a message
- How do we service these messages (i.e. call the callbacks)
- We can’t use system calls, need to manage this ourselves
- We have to decide what is acceptable
  - Latency
  - CPU overhead
  - Ordering

```
class RealTimeAsyncUpdater: RealTimeAsyncUpdateDispatcher : private Timer
{
public:
    RealTimeAsyncUpdateDispatcher()
    {
        startTimerHz (25);
    }

    void add (RealTimeAsyncUpdaterMessage&);
    void remove (RealTimeAsyncUpdaterMessage&);

private:
    void timerCallback() override
    {
        serviceUpdaters();
    }

    void serviceUpdaters();

    CriticalSection lock;
    Array<RealTimeAsyncUpdaterMessage*> updaters;
};
```

```
void RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher::add (RealTimeAsyncUpdaterMessage& m)
{
    const ScopedLock sl (lock);
    jassert (! updaters.contains (&m));
    updaters.add (&m);
}
```

```
void RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher::remove (RealTimeAsyncUpdaterMessage& m)
{
    const ScopedLock sl (lock);
    updaters.removeFirstMatchingValue (&m);
}
```

```
void RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher::serviceUpdaters()
{
    const ScopedLock sl (lock);

    for (auto updater : updaters)
        updater->serviceMessage();
}
```

```
    void serviceMessage()
    {
        if (shouldDeliver.compareAndSetBool (0, 1))
            owner.handleAsyncUpdate();
    }
```

juce::AsyncUpdater
Average = 20 microsecs, minimum = 5 microsecs, maximum = 102 microsecs

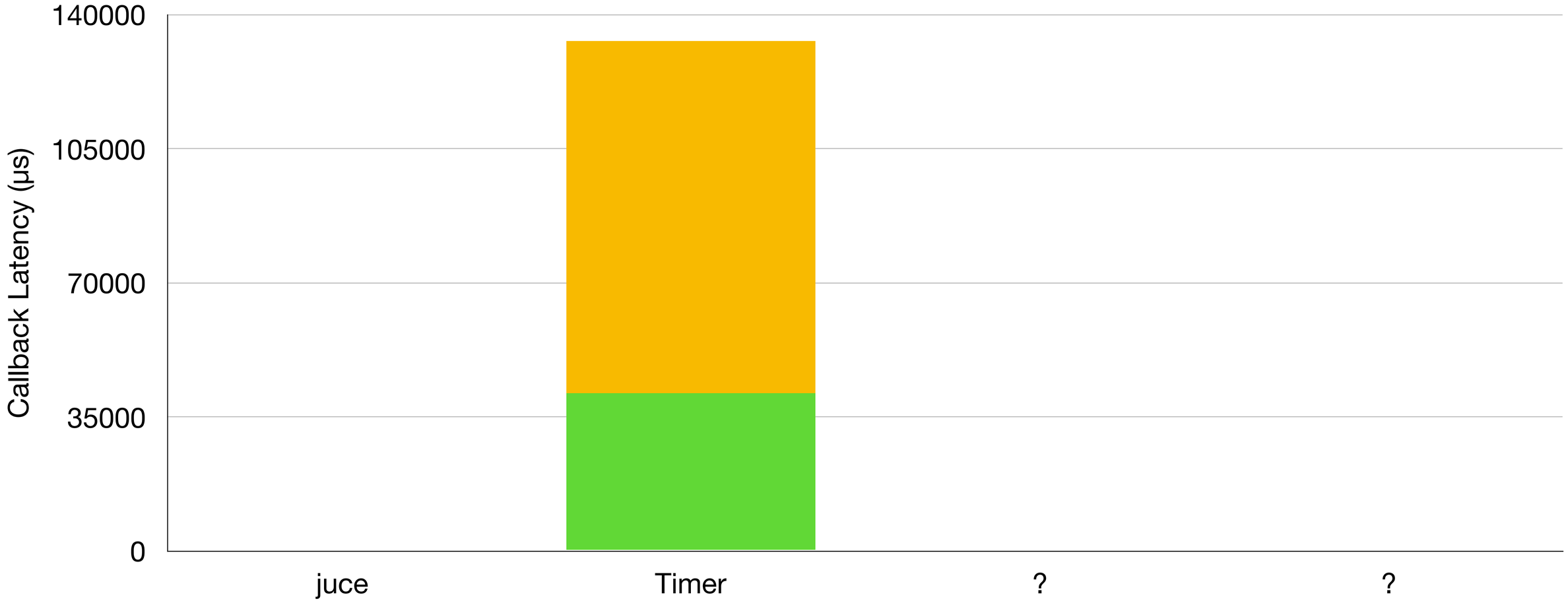
RealTimeAsyncUpdater (Timer Based)

```
Average = 41 millisecs, minimum = 39  millisecs, maximum = 42 millisecs, total = 40804 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 41 millisecs, total = 40813 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 41 millisecs, total = 40893 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 42 millisecs, total = 40817 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 41 millisecs, total = 40820 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 42 millisecs, total = 40834 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 42 millisecs, total = 40893 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 42 millisecs, total = 40844 millisecs
Average = 41 millisecs, minimum = 39  millisecs, maximum = 41 millisecs, total = 40859 millisecs
Average = 41 millisecs, minimum = 239 microsecs, maximum = 92 millisecs, total = 40989 millisecs
```

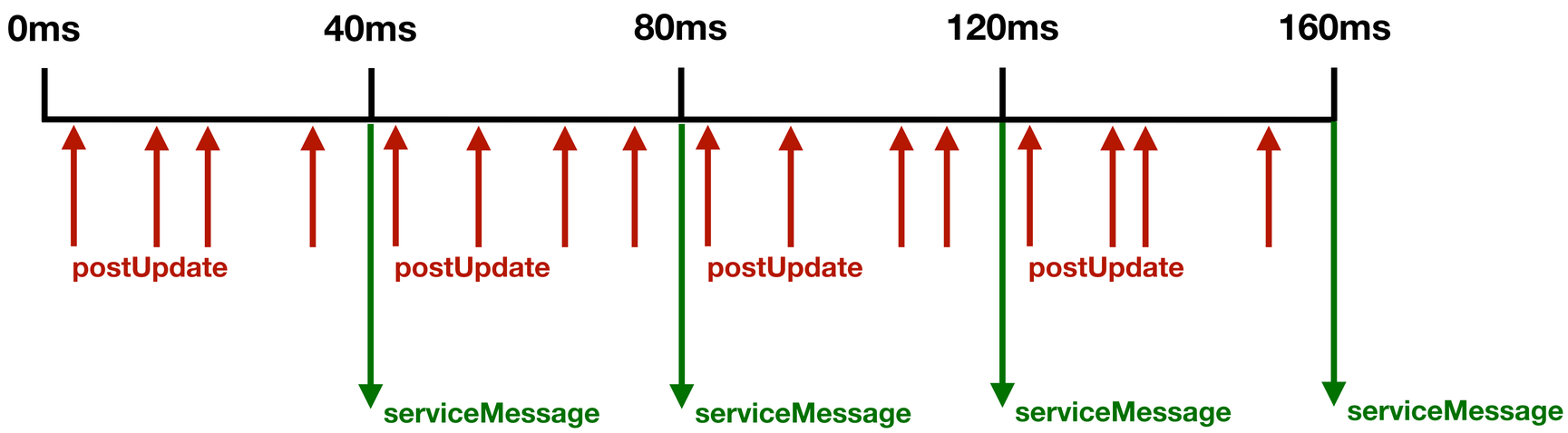
Average = 41 millisecs, minimum = 239 microsecs, maximum = 92 millisecs

Destroyed Latency

Round-trip Latency



```
    RealTimeAsyncUpdateDispatcher()
    {
        startTimerHz (25);
    }
```



Timer Solution

- Engineered a solution which:
  - Has an average latency of ~40ms with a jitter of~80ms 😞
  - CPU overhead of timer running 😐
  - Is real-time safe 😁
- Callbacks will happen in the order the **`RealTimeAsyncUpdaters`** are created


Can we do
better?

```
class RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher   : private HighResolutionTimer,
                                                             private AsyncUpdater

{
public:
    RealTimeAsyncUpdateDispatcher()
    {
        startTimer (5);
    }

    ~RealTimeAsyncUpdateDispatcher()
    {
        cancelPendingUpdate();
    }

    void add (RealTimeAsyncUpdaterMessage&);
    void remove (RealTimeAsyncUpdaterMessage&);

private:
    void hiResTimerCallback() override
    {
        triggerAsyncUpdate();
    }

    void handleAsyncUpdate() override
    {
        serviceUpdaters();
    }

    void serviceUpdaters();

    CriticalSection lock;
    Array<RealTimeAsyncUpdaterMessage*> updaters;
};
```

```
class RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher   : private HighResolutionTimer,
                                                              private AsyncUpdater

{
public:
    RealTimeAsyncUpdateDispatcher();
   ~RealTimeAsyncUpdateDispatcher();

    void add (RealTimeAsyncUpdaterMessage&);
    void remove (RealTimeAsyncUpdaterMessage&);

    void signal()
    {
        needsToService.store (true);
    }

private:
    void hiResTimerCallback() override
    {
        if (needsToService.exchange (false))
            triggerAsyncUpdate();
    }

    void handleAsyncUpdate() override
    {
        serviceUpdaters();
    }

    void serviceUpdaters();

    CriticalSection lock;
    Array<RealTimeAsyncUpdaterMessage*> updaters;
    std::atomic<bool> needsToService { false };
};
```

```
void RealTimeAsyncUpdaterMessage::postUpdate()
{
    shouldDeliver.compareAndSetBool (1, 0);
    dispatcher->signal();
}
```

juce::AsyncUpdater
Average = 20 microsecs, minimum = 5 microsecs, maximum = 102 microsecs

RealTimeAsyncUpdater (Timer Based)
Average = 41 millisecs, minimum = 239 microsecs, maximum = 92 millisecs

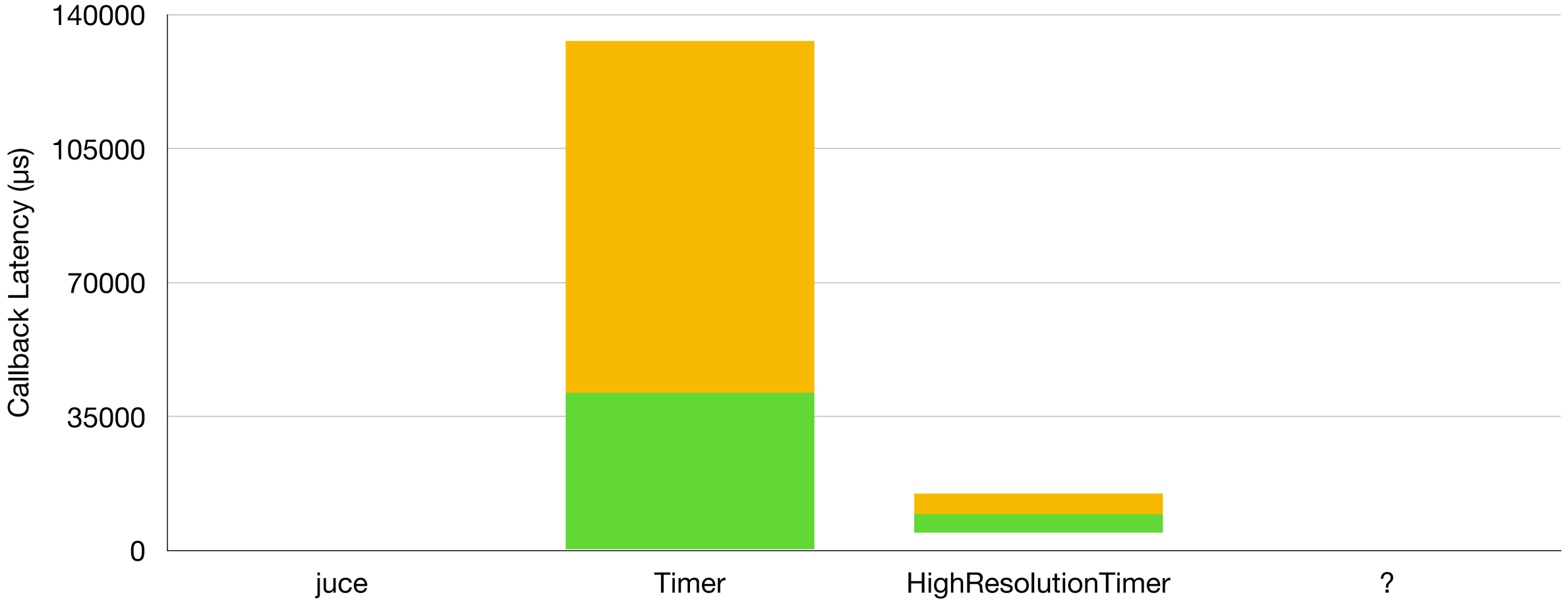
RealTimeAsyncUpdater (HighResolutionTimer Based)

```
Average = 5004 microsecs, minimum = 4760 microsecs, maximum = 5174 microsecs, total = 5004 millisecs
Average = 4997 microsecs, minimum = 4742 microsecs, maximum = 5216 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4806 microsecs, maximum = 5193 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4802 microsecs, maximum = 5178 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4759 microsecs, maximum = 5175 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4740 microsecs, maximum = 5204 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4643 microsecs, maximum = 5356 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4718 microsecs, maximum = 5230 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4793 microsecs, maximum = 5214 microsecs, total = 4997 millisecs
Average = 4997 microsecs, minimum = 4796 microsecs, maximum = 5211 microsecs, total = 4997 millisecs
```

Average = 4997 microsecs, minimum = 4643 microsecs, maximum = 5230 microsecs

Stable Latency

Round-trip Latency



HighResolutionTimer

- Engineered a solution which:
  - Will have an average latency of ~2.5ms with a jitter of~5ms 😞
  - Requires an additional thread to be running 😞
  - Relatively low CPU overhead 🙂
  - Is real-time safe 😁

Recap

- Created two solutions which are real-time safe
  - Return instantly from **`triggerAsyncUpdate()`**
- Both have additional overheads
- Both have worse average latency than **`juce::AsyncUpdater`**


Another
approach?

```
class RealTimeAsyncUpdater: RealTimeAsyncUpdateDispatcher  : private Thread,
                                                           private AsyncUpdater
{
public:
    RealTimeAsyncUpdateDispatcher()
        : Thread ("RealTimeAsyncUpdateDispatcher")
    {
        startThread();
    }

    ~RealTimeAsyncUpdateDispatcher()
    {
        cancelPendingUpdate();
        isDestructing = true;
        serviceEvent.signal();
        stopThread (10000);
    }

    void add (RealTimeAsyncUpdaterMessage&);
    void remove (RealTimeAsyncUpdaterMessage&);

    void signal()
    {
        serviceEvent.signal();
    }

private:
    void run() override
    {
        while (! threadShouldExit())
        {
            if (! isDestructing.load())
                serviceEvent.wait (-1);

            triggerAsyncUpdate();
        }
    }

    void handleAsyncUpdate() override
    {
        serviceUpdaters();
    }

    void serviceUpdaters();

    CriticalSection lock;
    Array<RealTimeAsyncUpdaterMessage*> updaters;
    WaitableEvent serviceEvent;
    std::atomic<bool> isDestructing { false };
};
```

```
void RealTimeAsyncUpdaterMessage::postUpdate()
{
    shouldDeliver.compareAndSetBool (1, 0);
    dispatcher->signal();
}
```

juce::AsyncUpdater
Average = 20 microsecs, minimum = 5 microsecs, maximum = 102 microsecs

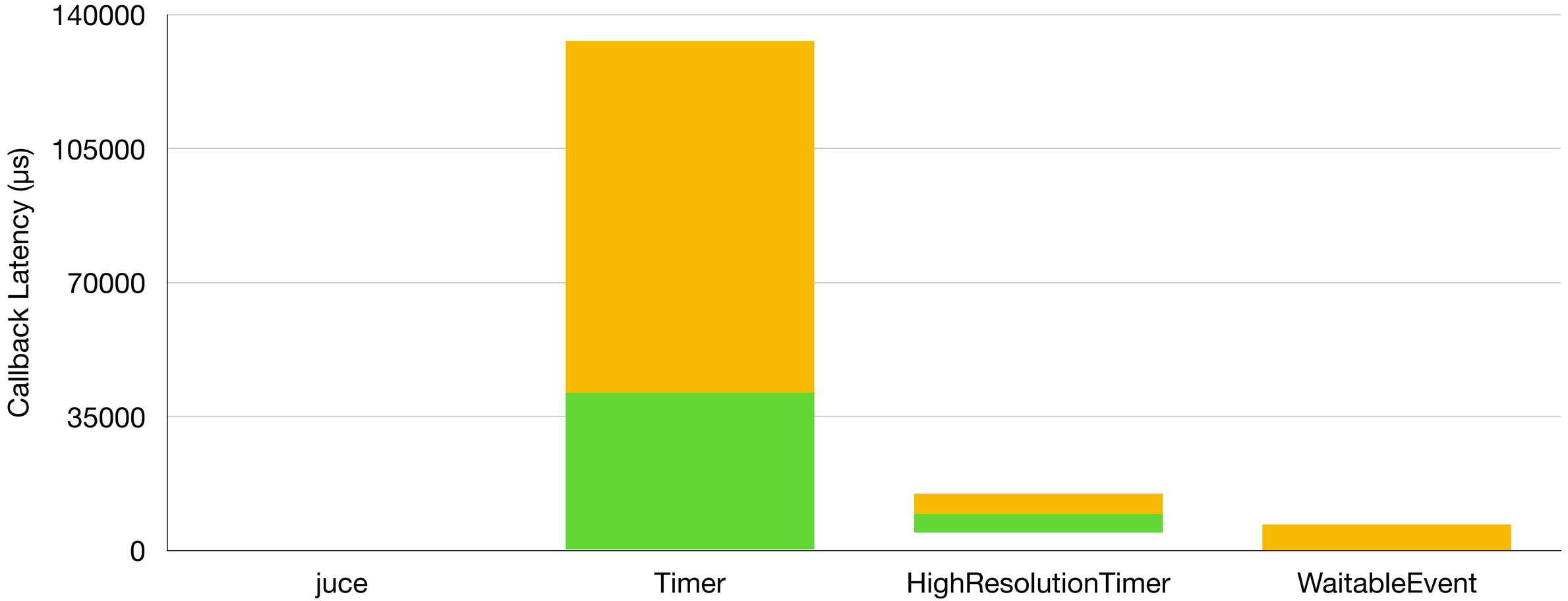
RealTimeAsyncUpdater (Timer Based)
Average = 41 millisecs, minimum = 239 microsecs, maximum = 92 millisecs

RealTimeAsyncUpdater (HighResolutionTimer Based)
Average = 4997 microsecs, minimum = 4643 microsecs, maximum = 5230 microsecs

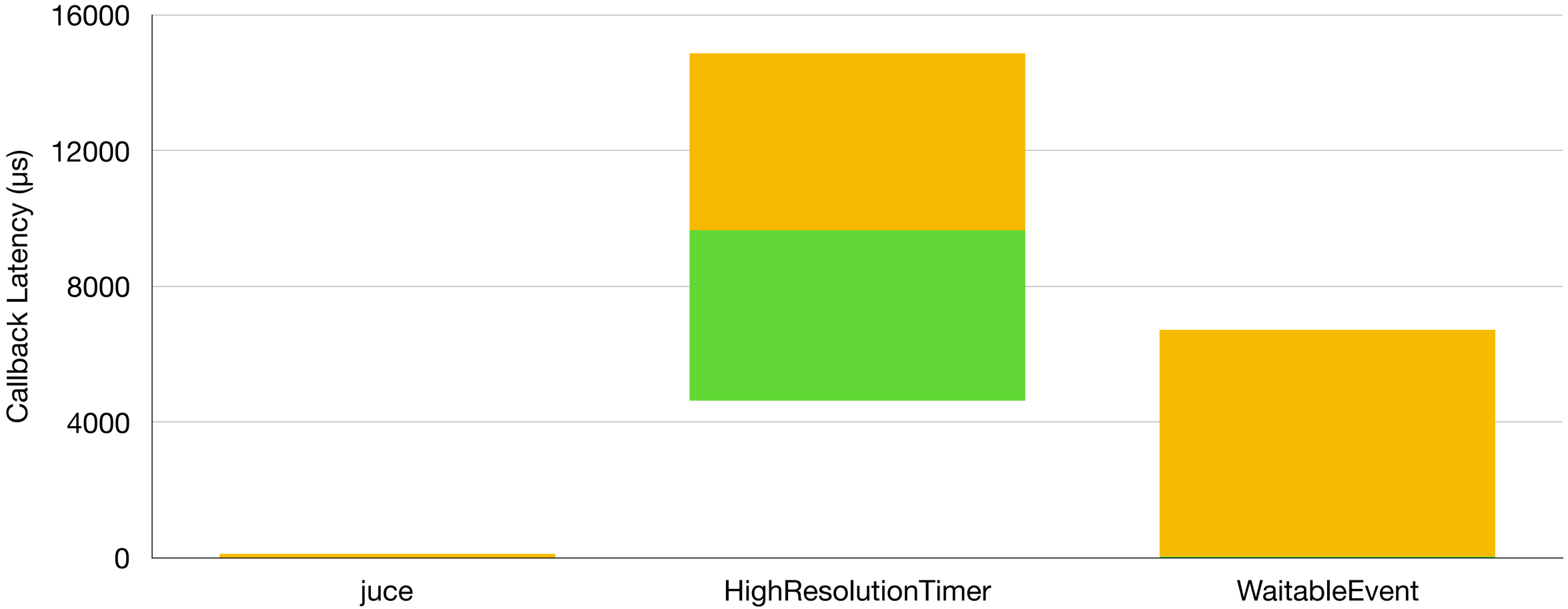
RealTimeAsyncUpdater (WaitableEvent Based)
Average = 16 microsecs, minimum = 7 microsecs, maximum = 64 microsecs, total = 16 millisecs
Average = 27 microsecs, minimum = 8 microsecs, maximum = 6695 microsecs, total = 27 millisecs
Average = 27 microsecs, minimum = 8 microsecs, maximum = 4673 microsecs, total = 27 millisecs
Average = 19 microsecs, minimum = 8 microsecs, maximum = 61 microsecs, total = 19 millisecs
Average = 20 microsecs, minimum = 8 microsecs, maximum = 96 microsecs, total = 20 millisecs
Average = 18 microsecs, minimum = 8 microsecs, maximum = 62 microsecs, total = 18 millisecs
Average = 16 microsecs, minimum = 7 microsecs, maximum = 48 microsecs, total = 16 millisecs
Average = 17 microsecs, minimum = 7 microsecs, maximum = 78 microsecs, total = 17 millisecs
Average = 16 microsecs, minimum = 7 microsecs, maximum = 42 microsecs, total = 16 millisecs
Average = 17 microsecs, minimum = 8 microsecs, maximum = 76 microsecs, total = 17 millisecs

Average = 20 microsecs, minimum = 7 microsecs, maximum = 6695 microsecs

Round-trip Latency



Round-trip Latency (Timer Removed)

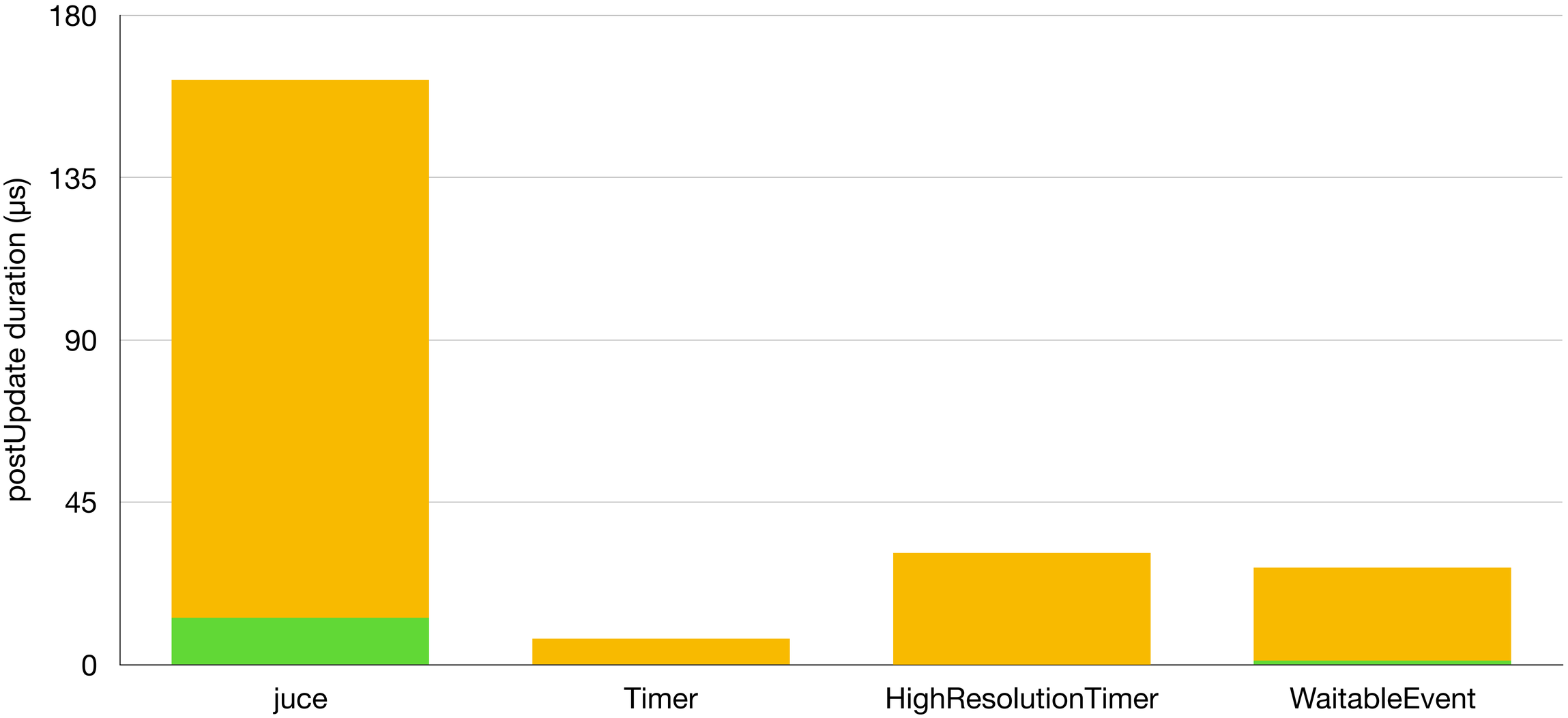


But wait...
`WaitableEvent`?

Real-time Safe?

- Engineered a solution which:
  - Is low latency (comparable to **`juce::AsyncUpdater`**) 😁
  - Low CPU overhead (additional **`notify_all`** cost) 😁
  - Therefore is not real-time safe 😞
  - Requires an additional thread to be running 😐
- This signal call will end update calling **`std::condition_variable::notify_all()`** which is a ***system call***
- Any system call which interacts with the the thread schedular ***could block***

Cost of `triggerAsyncUpdate()`



Summary

- Real-time vs. non-real-time guarantee
- Library-created background thread vs. timer overhead
- Minimum latency vs. jitter vs. cpu overhead
- Maybe this isn’t appropriate for a general-purpose library
- Think carefully about your use cases and trade-offs

More Problems

- Ordering of callbacks?
  - https://github.com/FigBug/Gin/blob/master/modules/gin/utilities/realtimeasyncupdater.cpp
  - Potential solution based on a lock-free queue?
- Template parameter for timer interval?

Real-time Trade-offs

David Rowland
@drowaudio

Questions?