Inter-Target Communication in Actor Framework

Allen C Smith, CLA
Consulting Software Architect

Linkedin.com/in/allencsmith

Allencsmith149.workfolio.com

niACS

NATIONAL INSTRUMENTS

ni.com

This presentation is copyrighted, 2017, by Allen C Smith.

About the author:

Allen C Smith has been using the NI platform to solve complex engineering challenges for over 23 years. He is a Certified LabVIEW Architect and Certified TestStand Developer. He has been an Alliance Partner, worked for Alliance Partners, and served as a Systems Engineer with National Instruments. While at NI, he developed the software tool support and official training course for the Actor Framework. During that time, he was known on the NI Forums as “niACS”.

Inter-Target Communication in Actor Framework

Allen C Smith, CLA
Consulting Software Architect

Linkedin.com/in/allencsmith

Allencsmith149.workfolio.com

~~niACS~~ justACS

NATIONAL INSTRUMENTS

ni.com

He is once again an Alliance Partner, offering services as a consulting software architect and engineer. He remains an AF evangelist and active member of the AF community.

Having left NI, he is no longer “niACS”. Now, he is “justACS”.

Options for Inter-Target Communication

- Background
- Linked Actor Trees (Client-Server Connections)
  - Nested Endpoints
- Single Inter-Target Actor Tree
  - Caller Endpoints
  - Launch Remote Actor

NATIONAL INSTRUMENTS

Nested endpoints are in current use.

To my knowledge, the community has paid scant attention to Caller Endpoints and Launch Remote Actor. One of the goals of this presentation is to change that.

Background

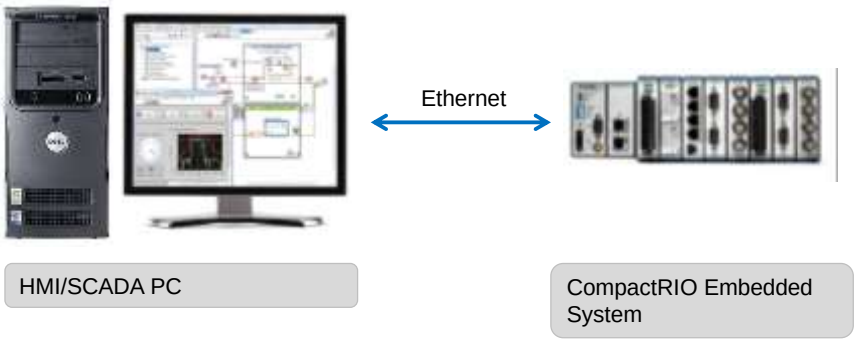
The Actor Framework

- Framework for building high concurrency systems
- Actors
  - parallel threads with data and functions
  - send and respond to messages
- Object-oriented framework
  - Build complex actors through inheritance
  - Build systems from sets of actors
- Common framework encourages reuse and collaboration

NATIONAL INSTRUMENTS

Community and collaboration are part of the promise of Actor Framework. A properly written actor is a discrete component that can easily be used in other actor systems. This lets us build an ecology where solutions to common problems become commonly available.

A Typical Multi-Target System

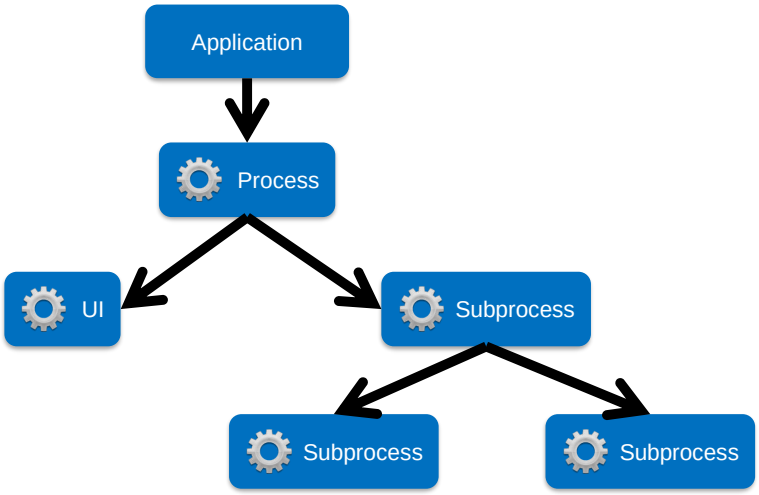


NATIONAL INSTRUMENTS

One such common problem is how to manage communication in a distributed, multi-target system.

This is what a typical distributed system looks like. You have a host pc (which may or may not be the dev. PC) and one or more distributed computing platforms. These can be any combination of cRIOs, PXI chassis, or other PCs. Generally, they are connected over a network.

Actor Framework on a Single Target

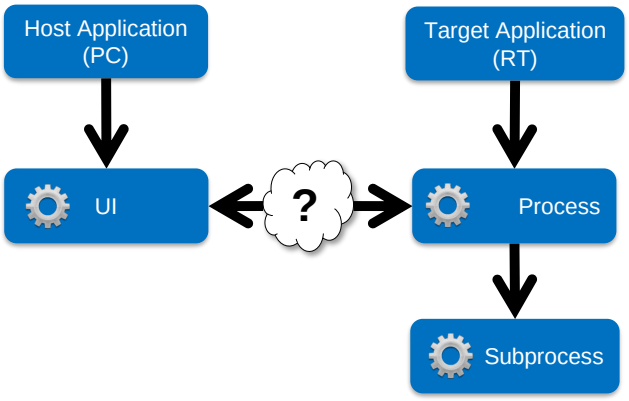


NATIONAL INSTRUMENTS

This is the recommended topology for Actor Framework systems. Your application launches a top level actor, which launches one or more subprocess actors. Those subproocesses may themselves launch subprocesses, and so on.

In this example, we have a user interface that represents the process to the user. The UI and process exchange data by enqueuing messages on their respective queues.

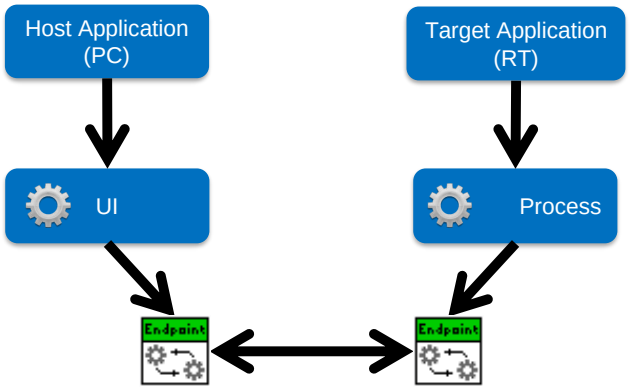
Linked Actor Trees on Two Targets



We have to change the topology if we want to distribute this system over two targets. We wind up with two actor trees running in parallel, and we need to build a way to exchange data between the UI and process.

LabVIEW queues don't work between instances of LabVIEW. So what can we do?

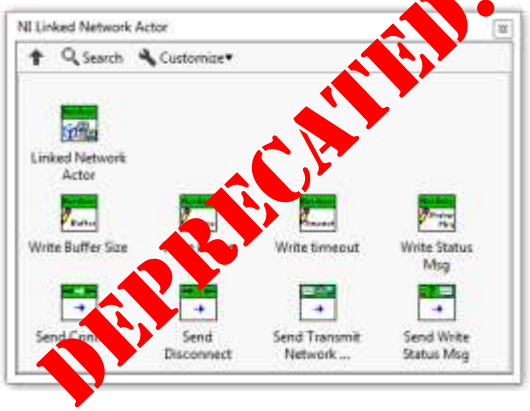
Linked Actor Trees



Here is one approach, which should look familiar to anyone who has worked with queued message handlers on distributed systems.

We create a new actor that manages one end of an inter-target communications protocol, and add an instance of this actor to each actor tree. Those endpoint actors broker the messasge traffic between UI and Process.

Linked Network Actor



- Bidirectional
- Uses Network Streams
- Persistent

- No TCP/IP
- Required a special “Transmit” message
- Cumbersome

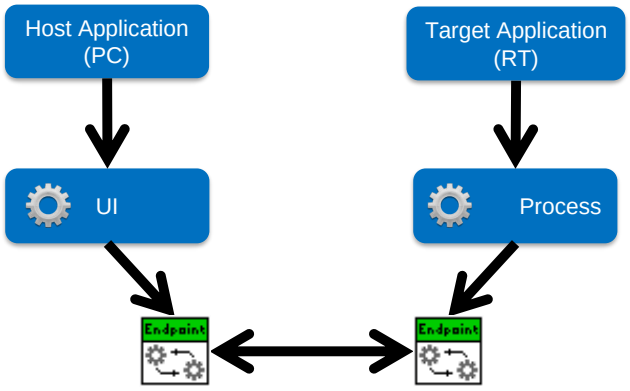
NATIONAL INSTRUMENTS

We released Actor Framework as a formal part of LabVIEW in 2012. Around the same time, I released the Linked Network Actor, a first attempt to address inter-target communication: the Linked Network Actor.

It was a successful release, but it became clear within a couple of years that something lighter and more flexible was needed. For these reasons, the LNA has been deprecated. It is still available on the forums, but no more work is being done on it.

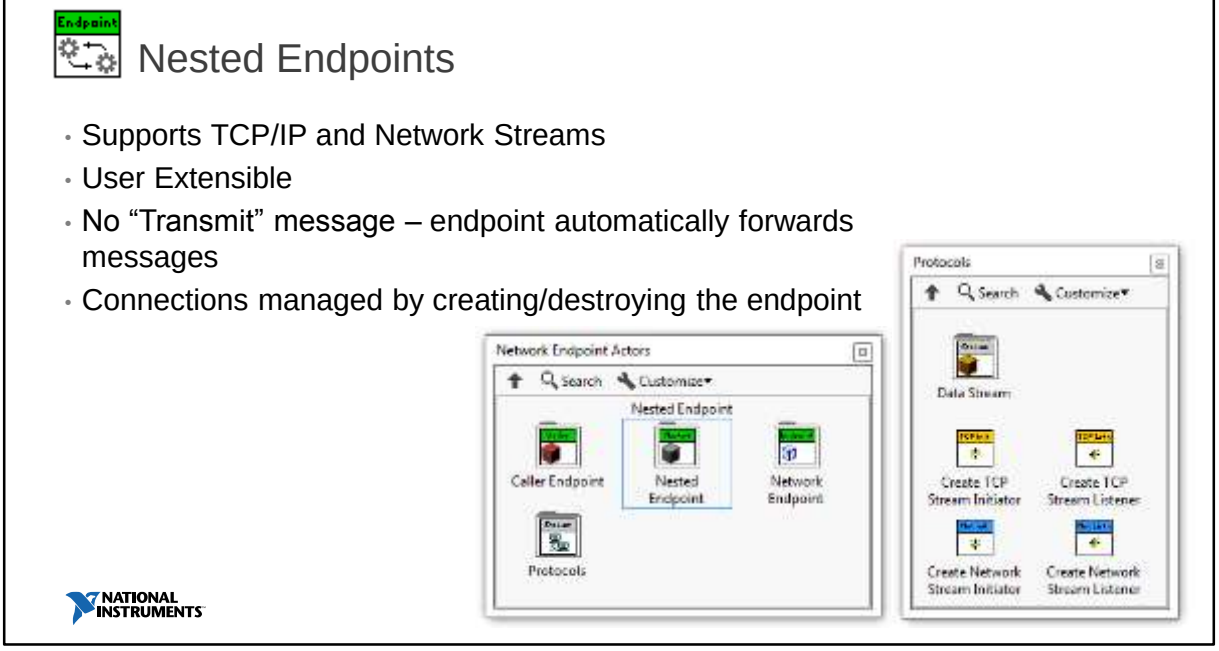
Client-Server Connections using
Nested Endpoint Actors

Linking Actor Trees with Nested Endpoints



NATIONAL INSTRUMENTS

Nested Endpoints replace the Linked Network Actor, and fulfil the same role.



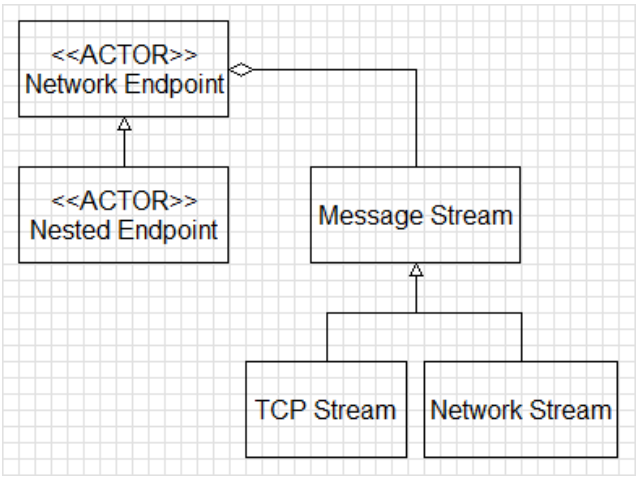
I first demonstrated Network Endpoints at the 2014 Americas CLA Summit, and then again at NIWeek 2014.

Sometime around 2013, we added the Receive Message override to AF. It’s not something you touch very often – most users never will – but it was added to support exactly this type of actor.

Receive Message lets us dispense with the “Transmit” message – you no longer have to package your message for shipping, you just send it to your endpoint, like any other message.

I also realized that we really didn’t need persistence, and that it added unnecessary complexity. It was a straightforward thing to make the actor attempt to connect at startup and disconnect when it receives a stop command, and doing so eliminated the Connect and Disconnect commands.

Nested Endpoints



NATIONAL INSTRUMENTS

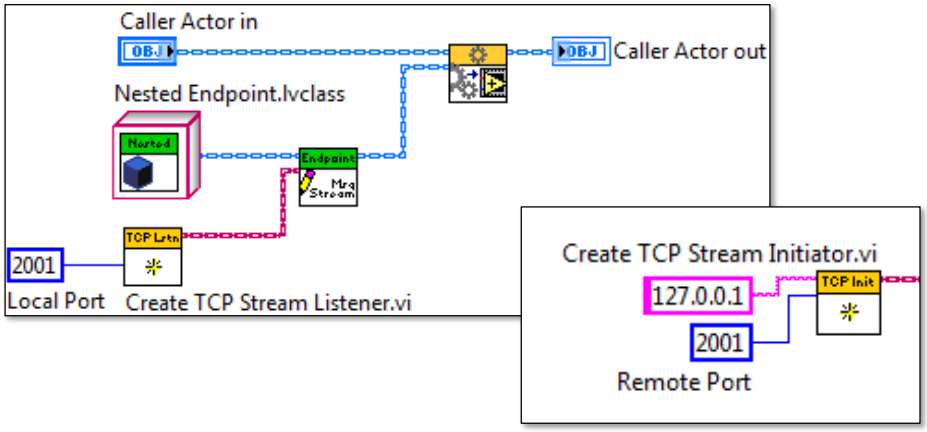
The UML for the Nested Endpoint is shown here.

Message Streams are the actual transmission protocols. Currently, two protocols are supported – Network Streams and TCP/IP. A message stream is just a standard LabVIEW class, not an actor.

The Network Endpoint is an actor that manages a message stream. It translates messages between actor space and the network.

Nested Endpoint is the implementation of Network Endpoint for our current use case, namely, to give network connectivity to some calling actor. There is another implementation, the Caller Endpoint, that we will discuss shortly.

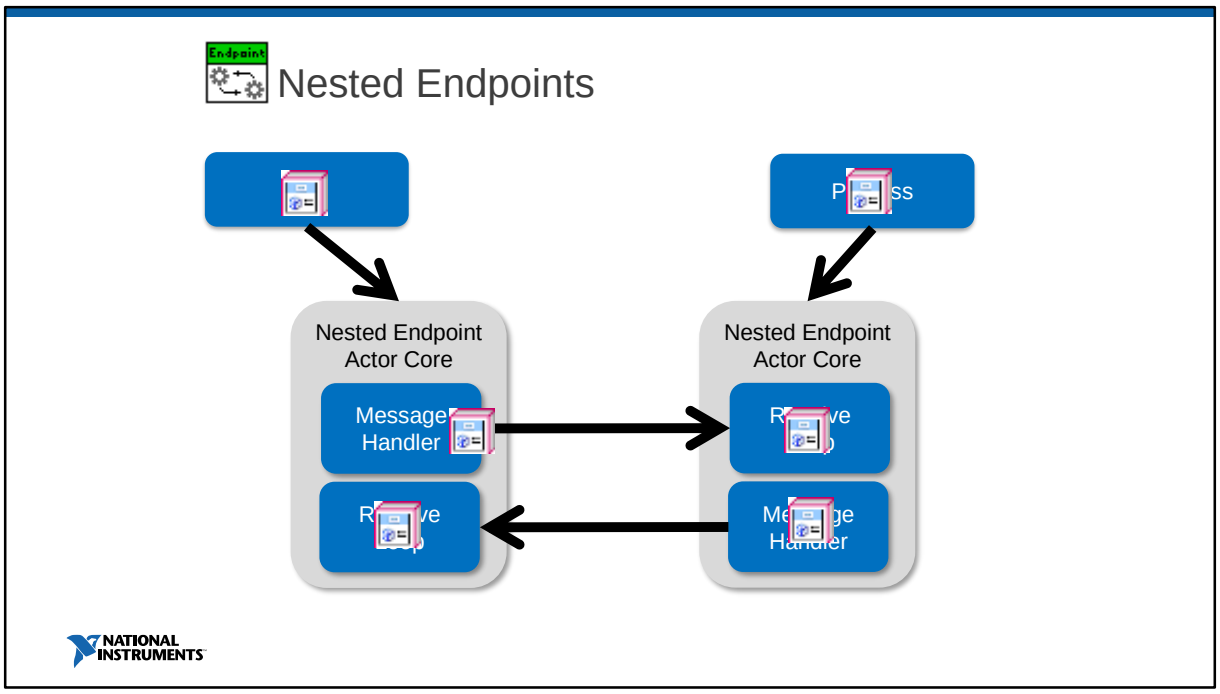
Nested Endpoints



NATIONAL INSTRUMENTS

As you saw in the Distributed Systems exercise, using a nested endpoint is straightforward. Assign your desired message stream to the endpoint, and launch it as you would any other actor.

The two implemented protocols, TCP/IP and Network Streams, come in two flavors – Listener and Initiator. Listeners wait passively for a connection, and are suitable for the server side of your application. **[Build]**: Initiators seek a Listener endpoint and establish a connection.



One of the actors in the Network Endpoints package is the Nested Endpoint. Like the Linked Network Actor, Nested Endpoints implement peer-to-peer messaging.

Some Messages Are Not Forwarded

Stop

Report Error

Last Ack

NATIONAL INSTRUMENTS

Now, as it turns out, there are a few messages we don't want to forward to the remote endpoint. So we've added some message filtering.

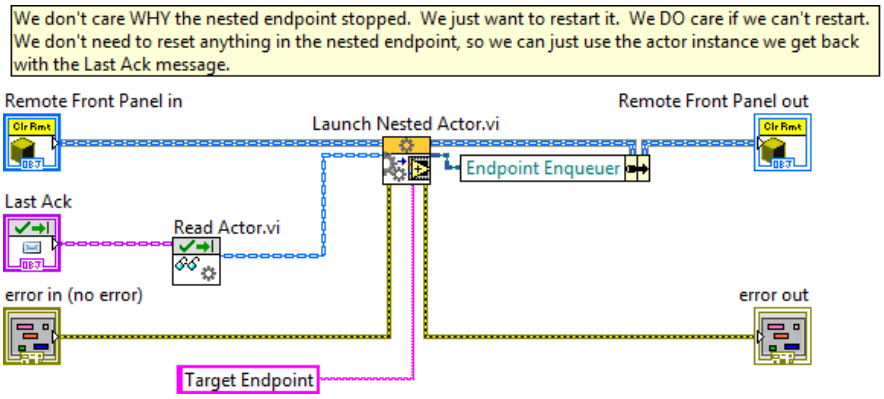
If the endpoint receives a Stop, Report Error, or Last Ack message on its own queue, then it handles the message normally.

This only makes sense. We want to handle the stop message locally, because that's where we will disconnect from the remote. If we forward a Stop to the remote, it will be passed to the Remote's caller, which is probably not desirable behavior.

Likewise, we want to be able to locally process Report Error messages from our own helper loop.

We also handle Last Acks locally, for reasons that will be clear in a bit.

But What About Persistence?

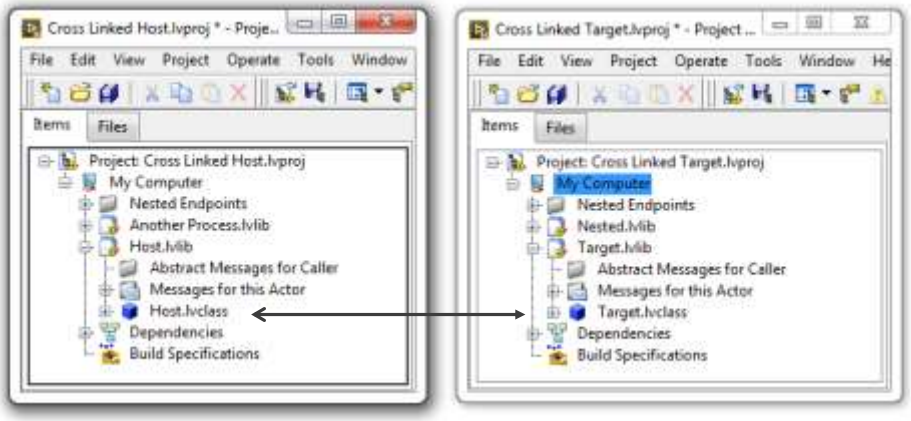


NATIONAL INSTRUMENTS

Generally, we'll want our listener to resume listening after a connection has been terminated. We can use Actor Framework's Last Ack functionality to implement this behavior.

When an actor stops, it sends a Last Ack to its caller. It is a trivial matter to have the caller restart its listener endpoint when it handles this message. The Last Ack message contains the complete final state of the sending actor, which, of course, is just an instance of that actor class. Since we don't need to change any attributes of the nested endpoint, we can just pass the returned nested endpoint instance to a new call to Launch Nested Actor.vi

But there is a catch…

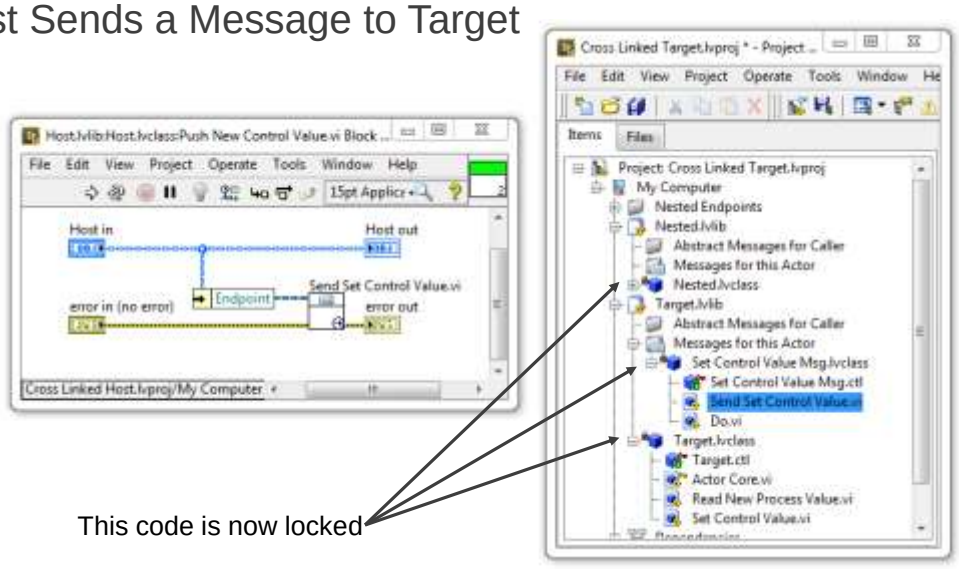


Target and Host Actors Exchange Messages

NATIONAL INSTRUMENTS

Your instructor will now present a short demonstration.

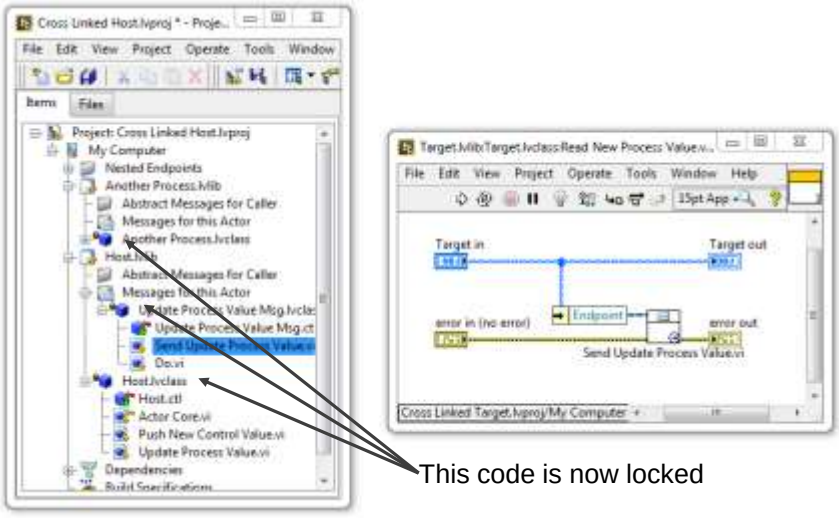
Host Sends a Message to Target



This code is now locked

NATIONAL INSTRUMENTS

Target Sends a Message to Host



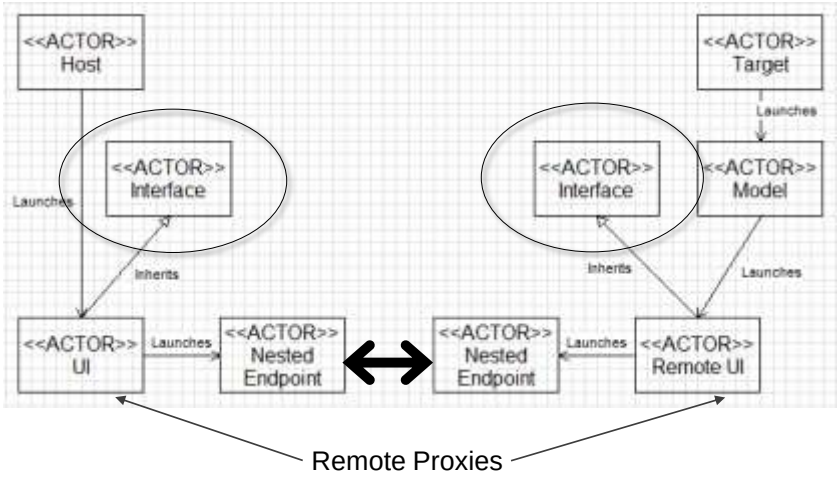
NATIONAL INSTRUMENTS

This is Intolerable

All the code is locked. And, as an added bonus, if your cross-linked actors call any target-specific code, your code is broken as well.

Let’s no do this.

Remote Proxies

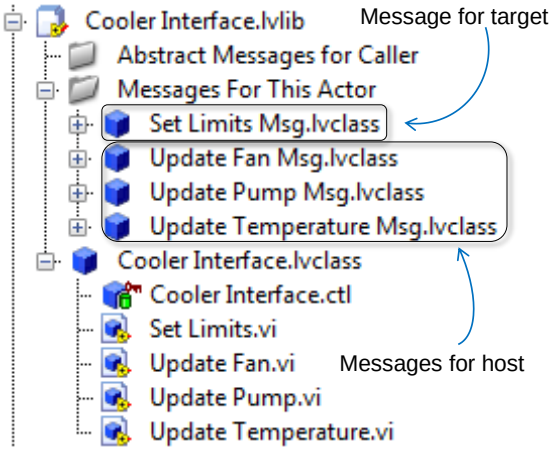


NATIONAL INSTRUMENTS

We introduce an abstract interface actor to our system. This proxy defines the communications interface between our two targets, but allows us to abstract away the specific implementations.

Interface Actor

- Defines an API (Low Coupling)
- Receives every message that crosses the application boundary
- Methods are dynamic dispatch



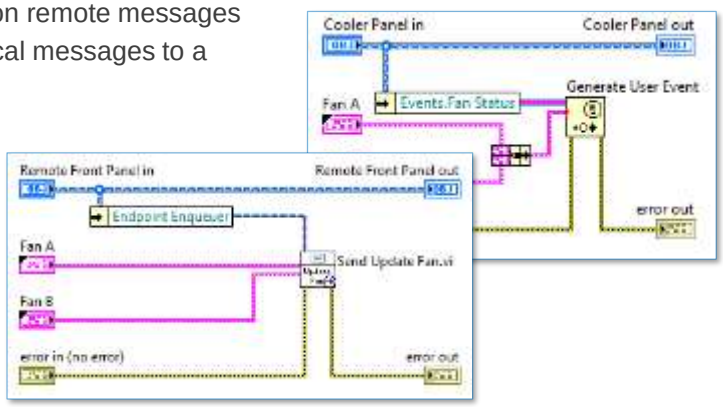
NATIONAL INSTRUMENTS

At first glance, it may seem that you will need a separate parent proxy class for the host and target, but it turns out that they can share the same class.

Both proxies, host and remote, must be able to handle the same set of messages, because all message traffic will pass through the proxy. The messages invoke dynamic dispatch methods that themselves contain no code.

Remote Proxy Actors

- Children of the Interface Actor
- Override parent methods to
  - act locally on remote messages
  - Forward local messages to a remote



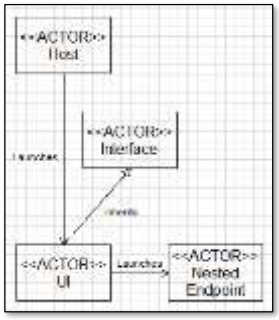
NATIONAL INSTRUMENTS

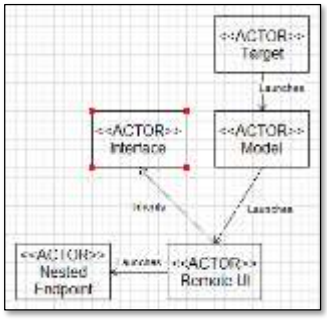
Each child proxy will do *something* with every message that must be sent over the connection. If the message is received from the remote, to be consumed by a local actor, the proxy must know to handle it or route it to another local actor. If the message comes from a local source, the proxy must know to forward it.

The children contain all of the local code, including any linked code or target-specific functions.

Remote Proxy Actors

- Use children of the interface
  - as the recipient (recipient *is a* proxy) or
  - as nested actor of the recipient (recipient *has a* proxy)

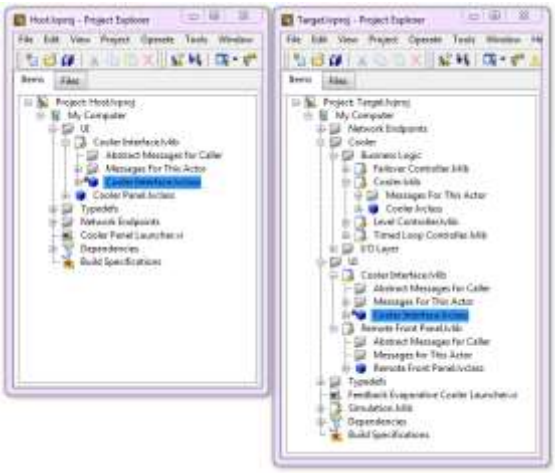




NATIONAL INSTRUMENTS

The targets each get their own implementation of the proxy parent. The child proxy can be used in one of two ways, as shown here.

Minimal Coupling



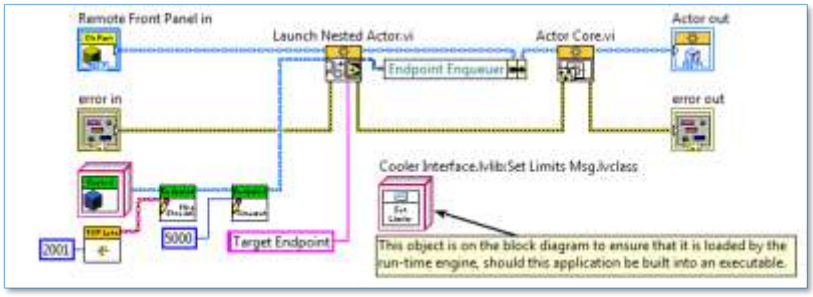
NATIONAL INSTRUMENTS

This solution minimizes decoupling because only the parent proxy and networking classes are loaded on both targets. Children of a class are only loaded when needed, and our local child proxies only call code that should be loaded on the same machine.

This kind of decoupling is truly powerful, and not just for resolving cross-platform loading issues. In the example here, Cooler Interface is the parent proxy. The children can be anything. In this case, the children manage network traffic and implement a remote UI. The next time we use the cooler, though, we may choose to run that code with a child of Cooler Interface that provides a local UI, or ties the cooler into another, larger system.

Guarantee Messages Are Loaded

Remote Front Panel:Actor Core

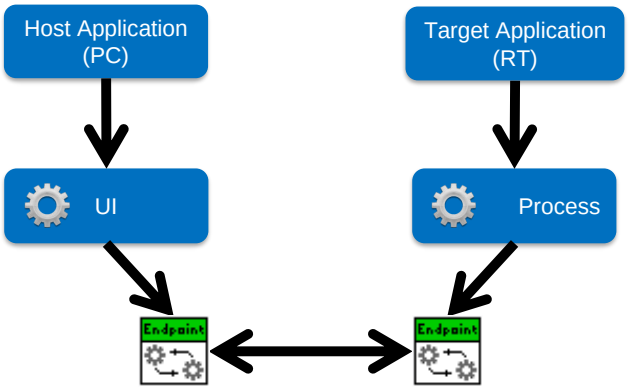


NATIONAL INSTRUMENTS

There is one last detail we need to consider. Typically, messages are only statically bound to the *sender*, which means they only get loaded into memory when the sender is loaded. When the sender is in another application instance, there is nothing that will automatically load those messages on the receiving side. So if a remote proxy gets a message from another application, it won't know how to handle it, and you'll get an error.

To solve this problem, simply add a constant for each of the proxy's messages to your block diagram somewhere. Actor Core is a tidy place to do this. This guarantees that the message classes will be loaded into memory with the remote proxy.

Linked Actor Trees

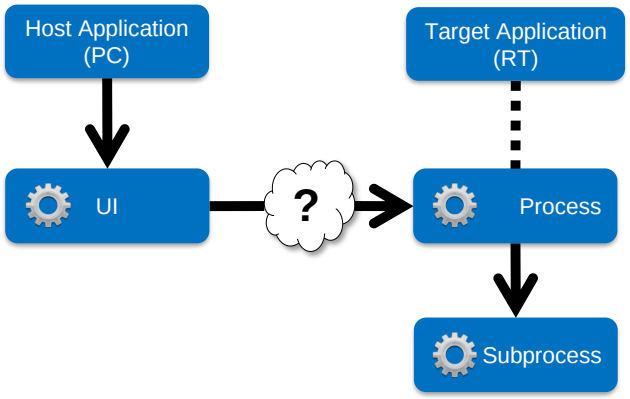


Connection functions as I/O

NATIONAL INSTRUMENTS

The model we've considered so far gives us two parallel actor trees, with a peer-to-peer connection between siblings at a low level. In this model, message traffic looks very much like I/O.

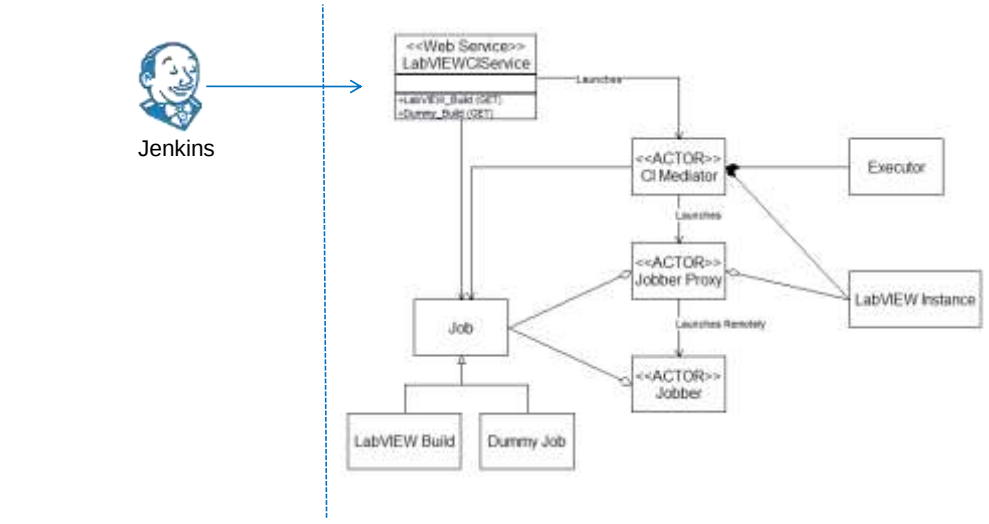
Can We Have a Single Tree?



NATIONAL INSTRUMENTS

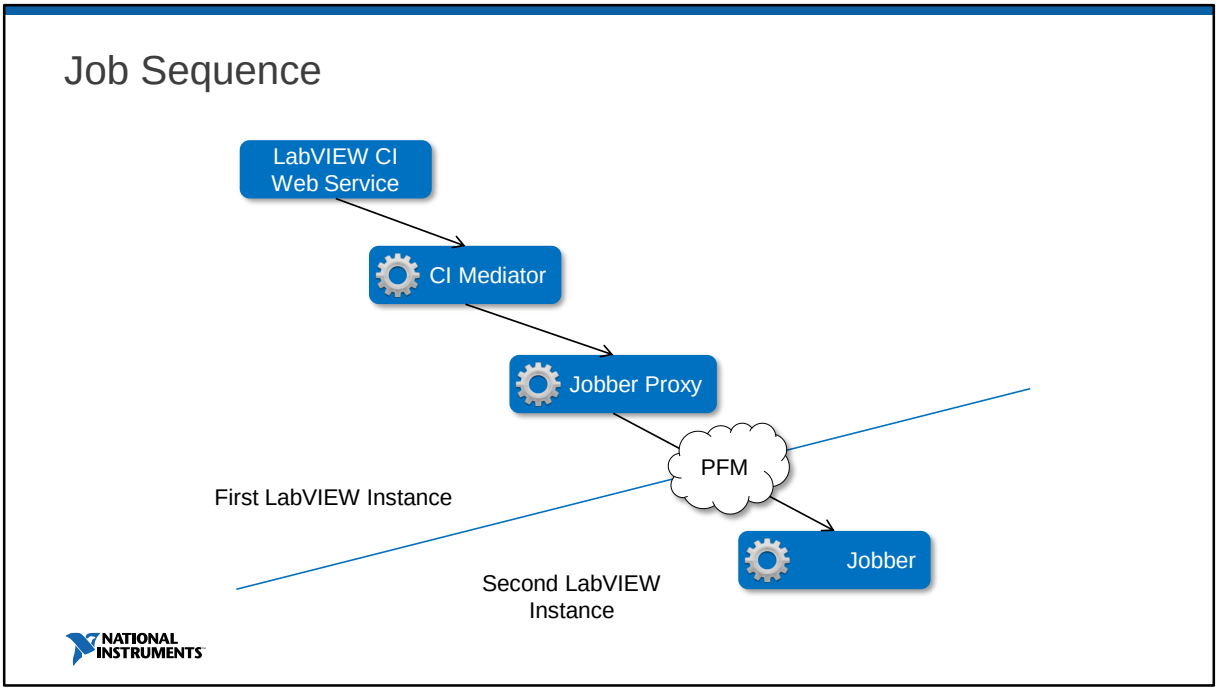
Linking actor trees solves a lot of use cases, especially those where the target application runs continuously, but the host application runs intermittently. But what if we want our application to look more like the single actor tree we started with?

Use Case: LabVIEW CI Web Service



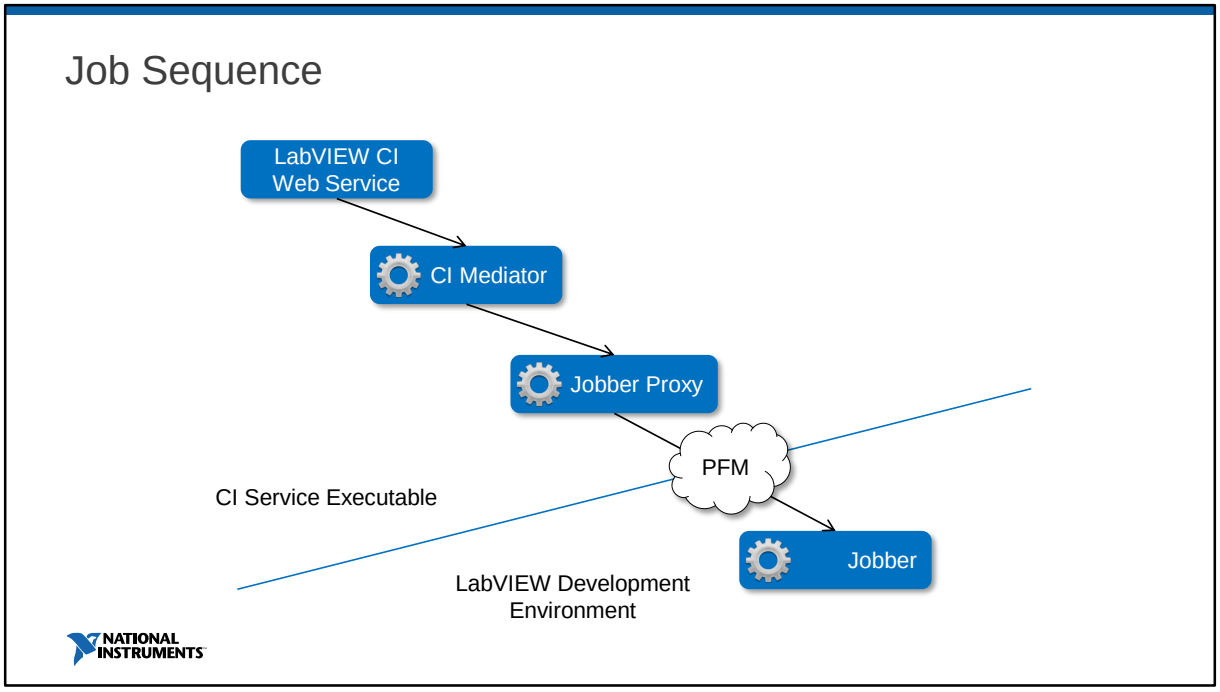
The LabVIEW CI Web Service, part of the LabVIEW Continuous Integration Project, is an example of an application that employs a single tree structure across multiple LabVIEW Instances. The Service is built with Actor Framework, to allow for future support for parallel tasks.

A Continuous Integration server that can act as an HTTP client can use this web service to test and build LabVIEW applications. To be a *continuous* service, we need to guarantee that the web service is always available, and it would be highly desirable if the web service were a built executable. But you can't invoke the Application Builder from an executable, and it is highly desirable that we close LabVIEW between each build or test step, to prevent any possibility of cross-linking between steps. This requires that we do the actual work in a fresh instance of the LabVIEW development environment.



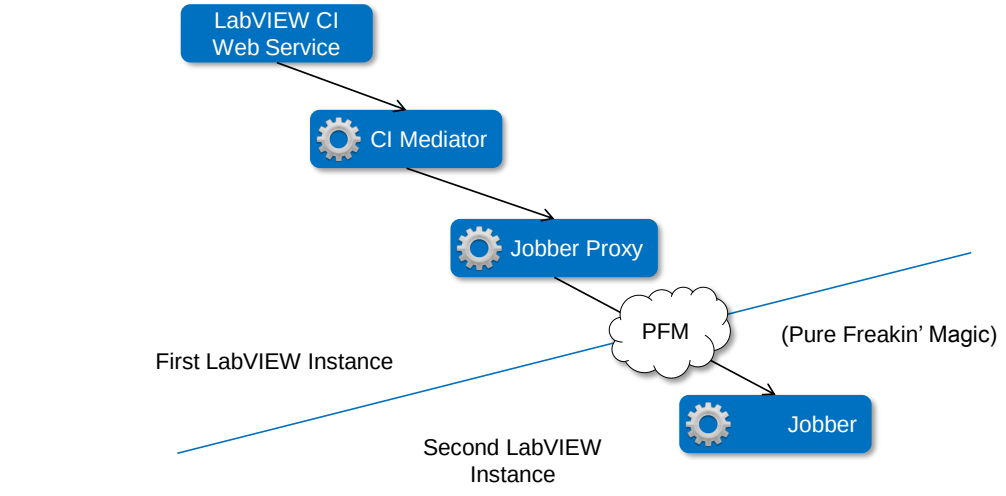
Here is the desired behavior. The actor system in our executable will launch an instance of LabVIEW, and then launch a nested actor in that instance to do the task…

Job Sequence



… and then roll it all up when the job is done.

Job Sequence

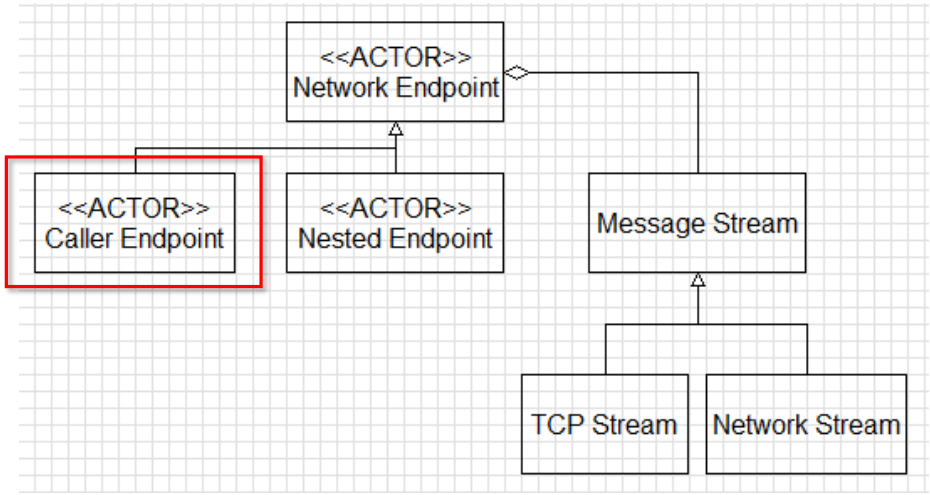


NATIONAL INSTRUMENTS

So how do we make this work?

Single Inter-Target Actor Trees using
Caller Endpoint Actors

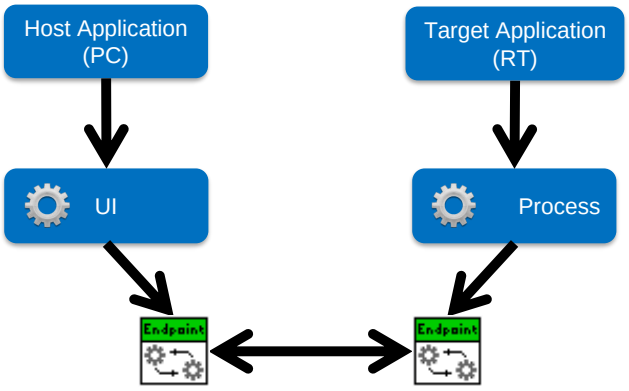
Caller Endpoints



NATIONAL INSTRUMENTS

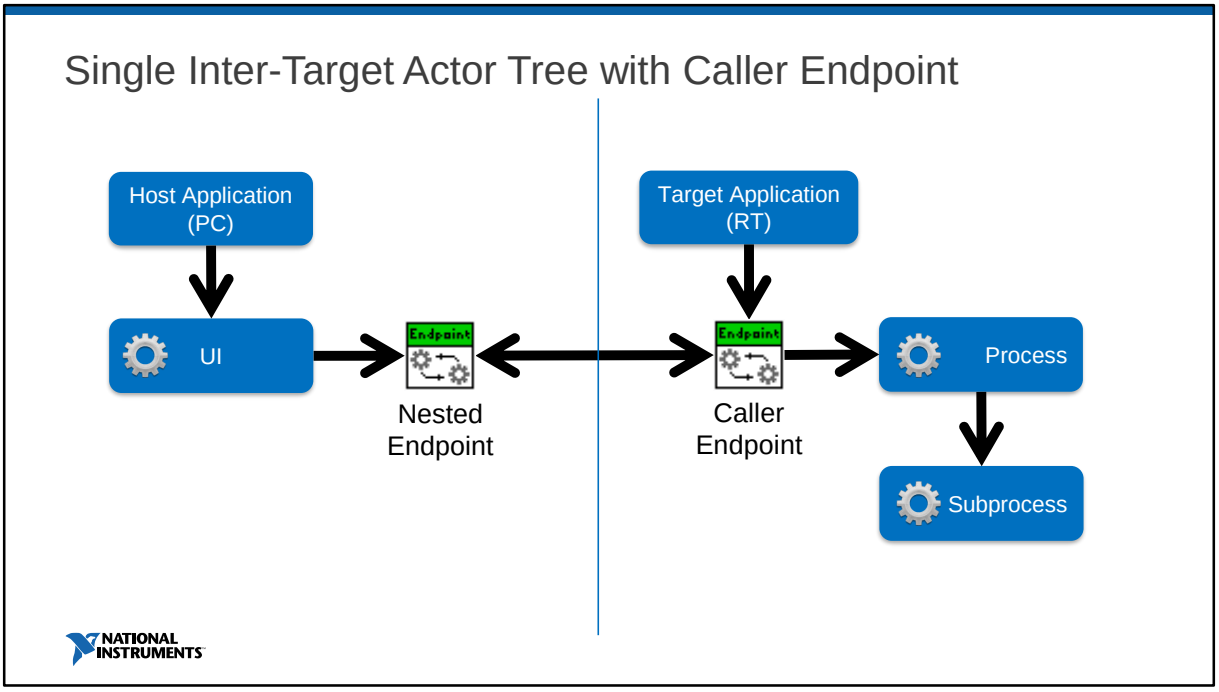
Caller Endpoints are another implementation of the Network Endpoint. As such, they use the same message stream classes, and they manage message traffic in the same way as Nested Endpoints. The difference is where they sit in the actor hierarchy.

Linked Actor Trees with Nested Endpoints



NATIONAL INSTRUMENTS

This is our model so far. Two actor trees launch in parallel, and the connection occurs toward the end of the process.

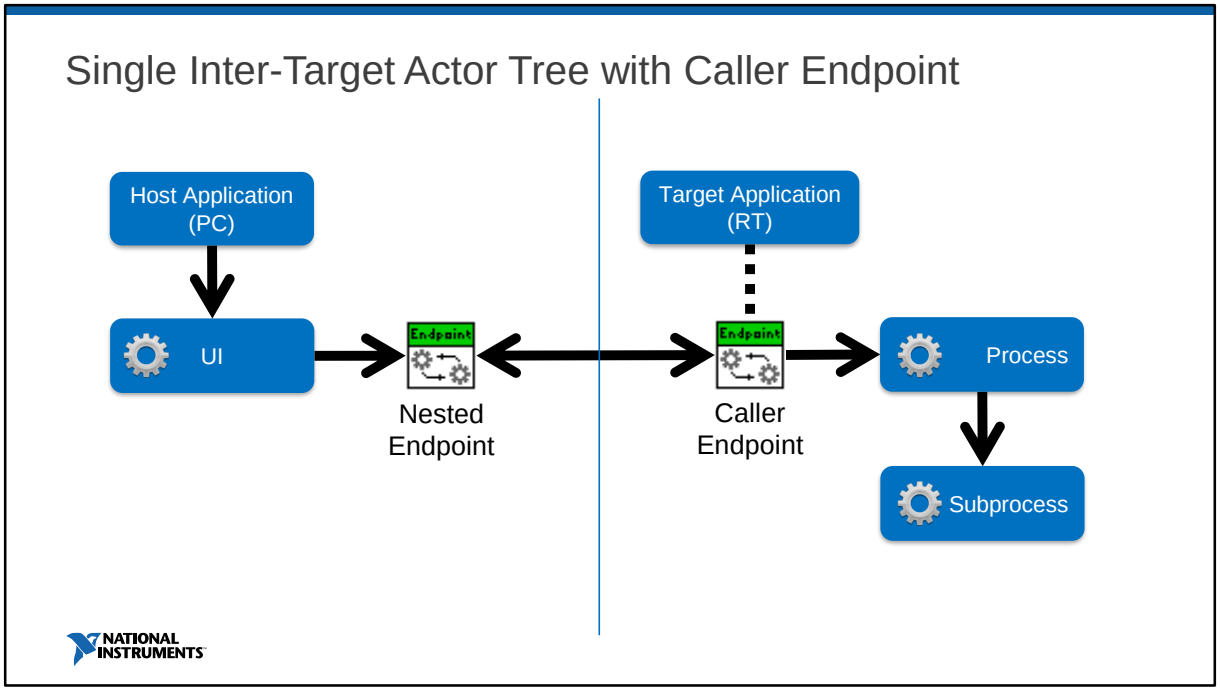


Caller Endpoints work differently. In this case, establishing the network connection is part of launching the actor tree.

A small target application is always running. It maintains a caller endpoint that is waiting for a connection.
[Build]: The host application launches its actor, in this case, a UI
[Build]: The UI initiates a connection by launching a Nested Endpoint.
[Build]: The Nested Endpoint establishes a connection with the Caller Endpoint on the remote target.
[Build]: Once a connection has been established, the Caller Endpoint launches its processes

If the connection is lost, the Caller Endpoint will stop its nested actors and then stop itself.

If the Caller Endpoint receives a Last Ack, it will forward it to its remote endpoint. That Nested Endpoint will forward the Last Ack to its caller and then Stop. TODO: There is a bug; currently no on stops on Last Ack. Fix this before NIWeek.

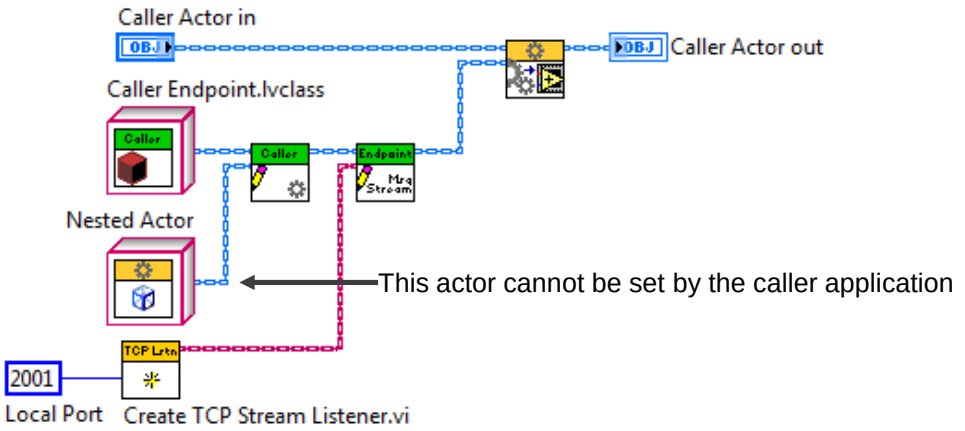


If a Caller Endpoint receives a message on its queue, it forwards that message to its remote counterpart. If a Caller Endpoint receives a message from its remote counterpart, it passes that message down to its own nested actors. In fact, the only message the Target Application will ever receive from its Caller Endpoint is a Last Ack message when it stops.

If the connection is lost, the Caller Endpoint will stop its nested actors and then stop itself.

If the Caller Endpoint receives a Last Ack from its nested actor, it will forward it to its remote endpoint. That Nested Endpoint will forward the Last Ack to its caller and then Stop. TODO: There is a bug; currently no on stops on Last Ack. Fix this before NIWeek.

Caller Endpoints

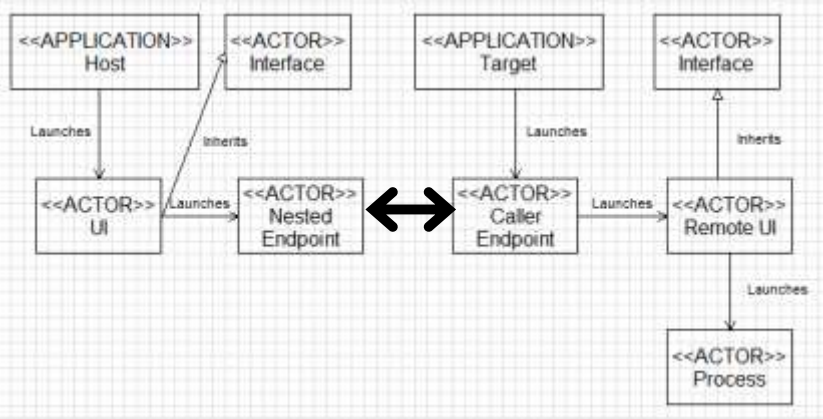


NATIONAL INSTRUMENTS

Use them much like you use Nested Endpoints, except you have to specify the actor to launch. Note that the actor is set on the target side. If you need the host to be able to specify the actor at run time, you'll need to do some extra work.

TODO: for the compare/contrast slide, note that the actor to be launched is fixed at edit time. Note also that TCP can be secured and plays better with firewalls; VI Server is not/does not.

Minimizing Coupling



NATIONAL INSTRUMENTS

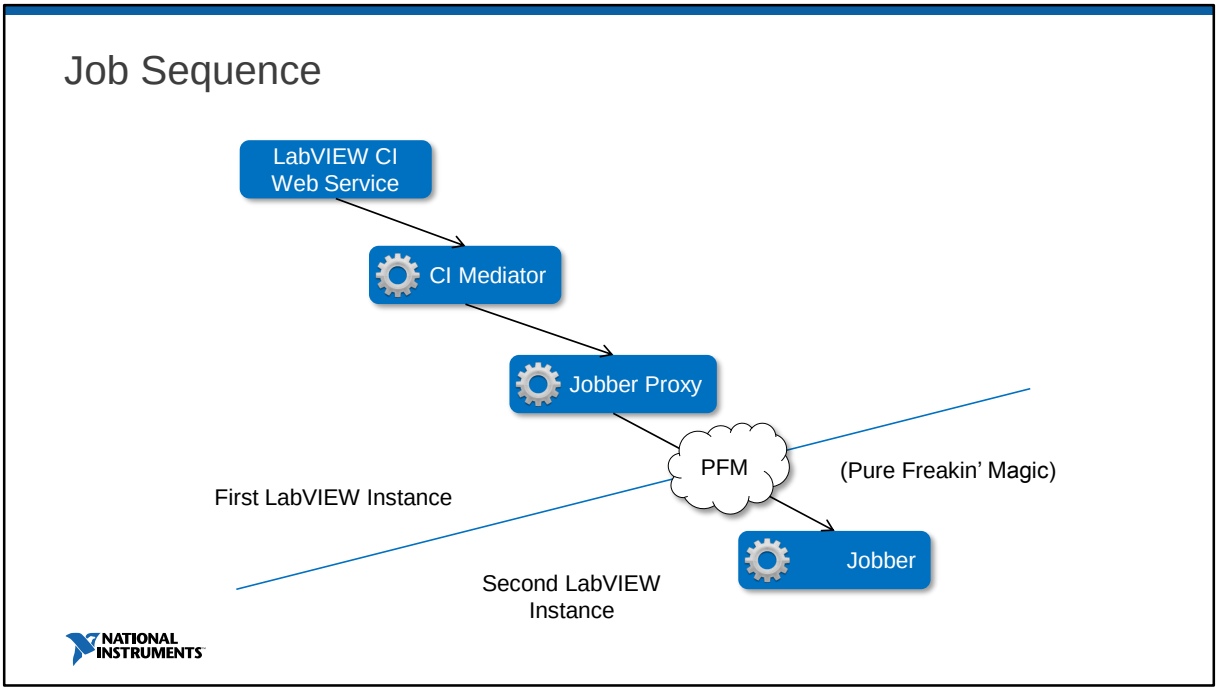
A minimal coupling solution for Caller Endpoints might look like this. You'll still need a proxy actor that defines all the message traffic, with children of those proxies implementing target specific code.

As it turns out, I have done very little work with Caller Endpoints. For my use case, they were pre-empted by another solution that proved to be a better fit.

Single Inter-Target Actor Trees using Launch Remote Actor

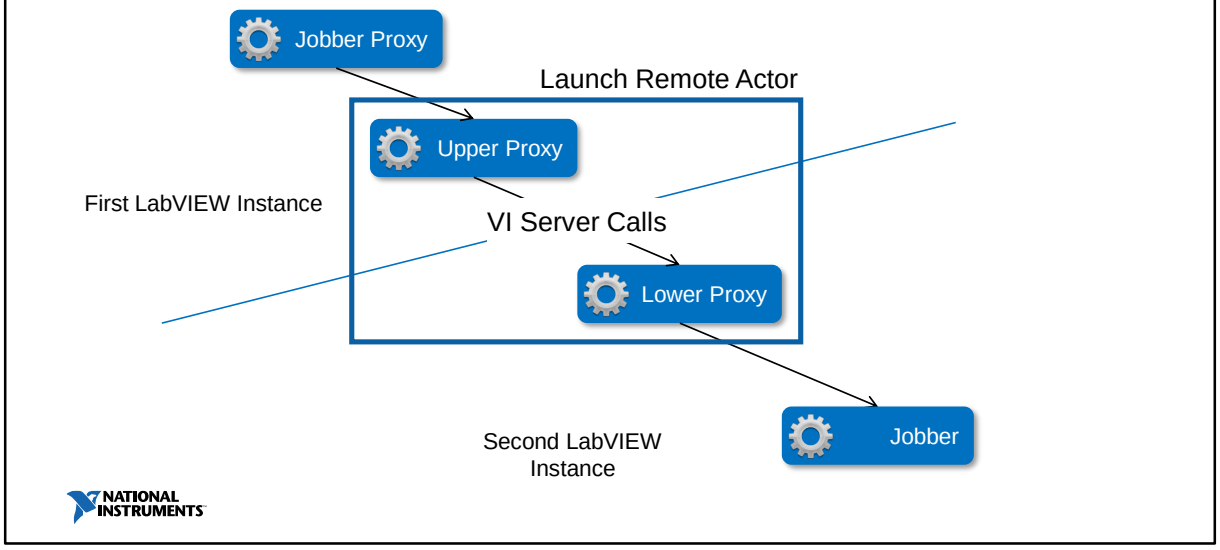
Launch Remote Actor is why so little has been done with Caller Endpoints. Stephen Loftus-Mercer developed these, and first released them in January, 2013

Launch Remote Actor has been around for a while, but it originally required an experimental fork of the Actor Framework. Around the time I was developing Caller Endpoints, some of the required changes were made to the shipping product, and we realized that we could change the LRA to depend only on the shipping version of AF.

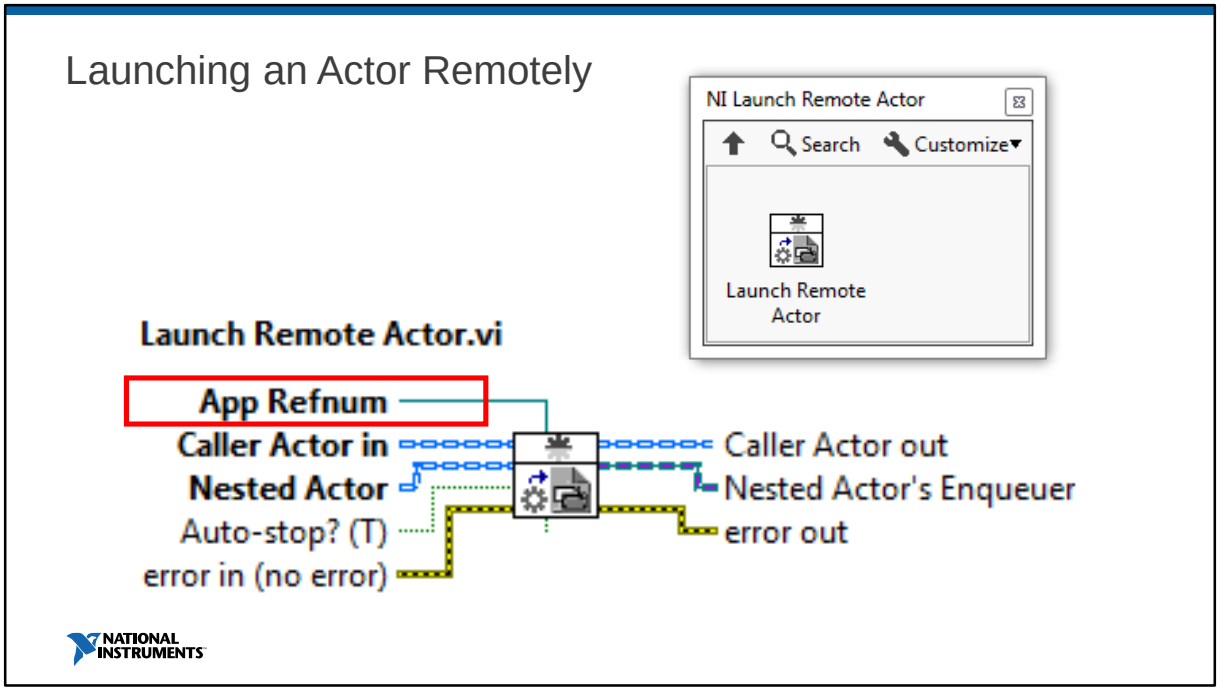


Launch Remote Actor answers the same need as Caller Endpoints. But Launch Remote Actor has a much cleaner interface.

Launching an Actor Remotely



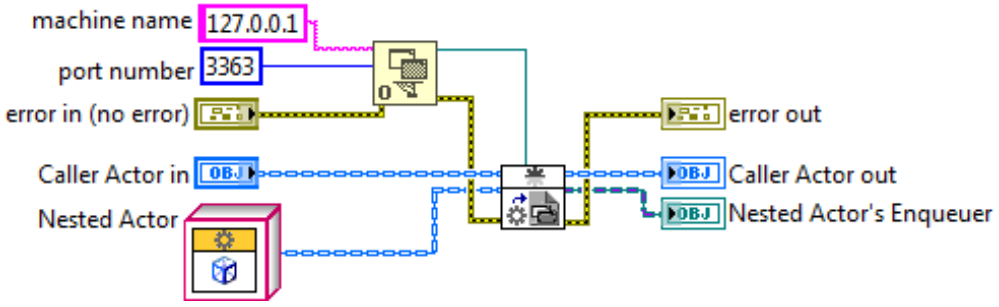
The reality, of course, is a bit more complicated. Launch Remote Actor.vi fronts for an entire chain of events that establishes the connection between Caller and Nested. The actual inter-target exchange is brokered by a pair of actors, Upper Proxy Actor and Lower Proxy Actor.



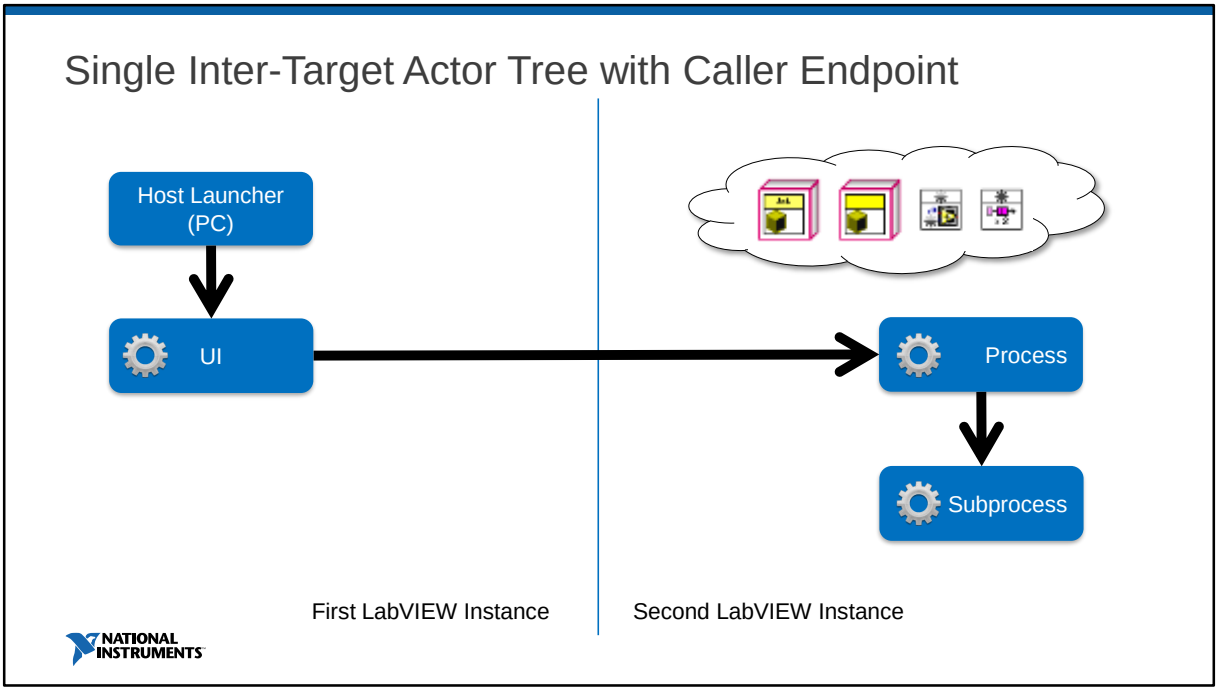
The API for launching a remote actor is very simple – there is only one VI. Launch Remote Actor.vi works almost identically to Launch Nested Actor. You pass in the actor you want to run, and you get back a valid enqueuer for that actor.

There is one crucial difference. The VI takes as input a reference to the application instance where the nested actor will run.

Using Launch Remote Actor



But again, all of the preceding information is background. From the user's perspective, this is sufficient to launch a remote actor.



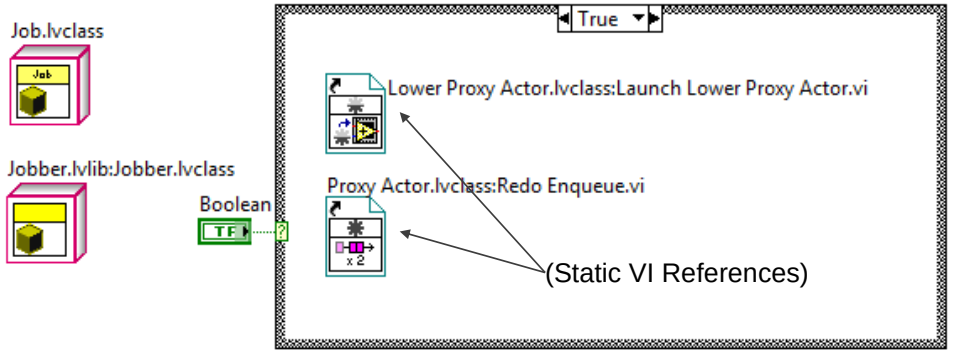
Caller Endpoints work differently. In this case, establishing the network connection is part of launching the actor tree.

A small target application is always running. It maintains a caller endpoint that is waiting for a connection.
[Build]: The host application launches its actor, in this case, a UI
[Build]: The UI initiates a connection by launching a Nested Endpoint.
[Build]: The Nested Endpoint establishes a connection with the Caller Endpoint on the remote target.
[Build]: Once a connection has been established, the Caller Endpoint launches its processes

If the connection is lost, the Caller Endpoint will stop its nested actors and then stop itself.

If the Caller Endpoint receives a Last Ack, it will forward it to its remote endpoint. That Nested Endpoint will forward the Last Ack to its caller and then Stop. TODO: There is a bug; currently no on stops on Last Ack. Fix this before NIWeek.

Considerations for the Target Application



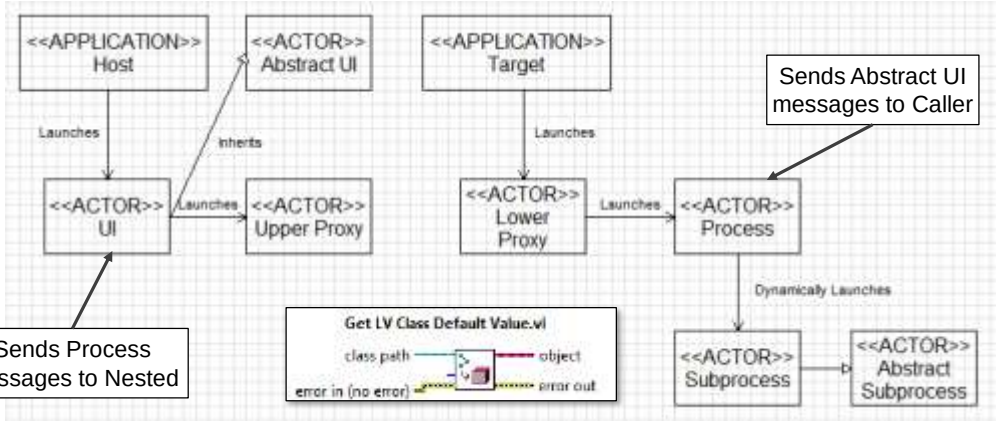
NATIONAL INSTRUMENTS

As with the Network Endpoints, you will need to ensure that several VIs and classes are available in memory. But the list is a bit longer.

In addition to any message class that it will receive from its remote Caller, the application where the Nested actor will run must also keep Launch Lower Proxy Actor.vi, Redo Enqueue.vi, and the Nested Actor to be launched in memory. Any classes to be sent as message data should also be present.

Note that this is true if you whether you are using the development environment or an executable – you'll need to have a VI that is open that has all these things. In the dev environment, it suffices to open a VI that contains the desired code. An executable will need to run that VI, and keep it running for as long as you need to launch a nested actor.

Minimizing Coupling



NATIONAL INSTRUMENTS

Decoupling gets a little more complex, because we can't replace the Process actor with an abstract proxy class. The UI actor is statically linked to the actual Process actor to be run on the target.

The benefit is that we don't need a proxy for the Process actor – the Caller has the API of the Process actor. But if the Subprocess actor contains any target-specific code, we'll need to load its class constant dynamically, and we'll need an abstract parent class for it.

We'll still need an abstract parent actor for the UI, if it contains any host-specific code. Process will use this API to send messages to its caller.

Caller Endpoints or Launch Remote Actor?

Caller Endpoints	Launch Remote Actor
Easier Code Decoupling	Simpler API
TCP/IP, Network Streams, User Defined	VI Server
Nested Actor Specified by Lower Target at Edit Time	Nested Actor Specified by Upper Target at Run Time

NATIONAL INSTRUMENTS

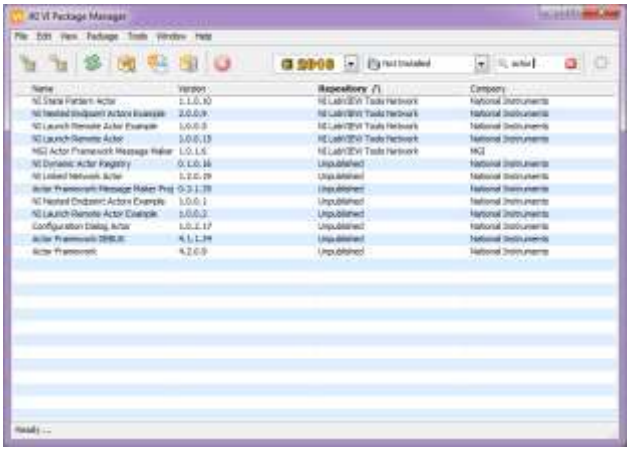
Linked Actor Trees or Single Inter-Target Tree?

Linked Trees	Inter-Target Tree
Nested Endpoints	Caller Endpoints or Launch Remote Actor
Client-Server Architecture	Distributed Actor Hierarchy
Client Connects Intermittently	Caller Determines Life Cycle, Remote Nested May Be Intermittent

NATIONAL INSTRUMENTS

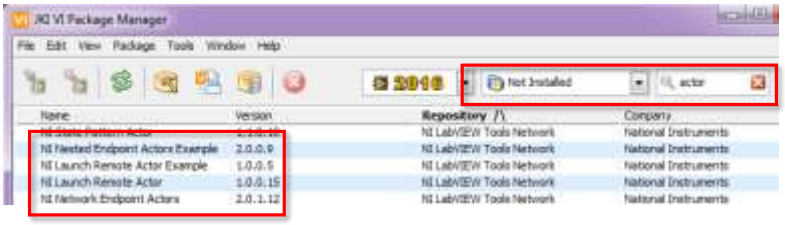
Where Can I Find Them?

Available on the LabVIEW Tools Network



These packages are available on the LabVIEW Tools Network, and you can download and install them with VI Package Manager.

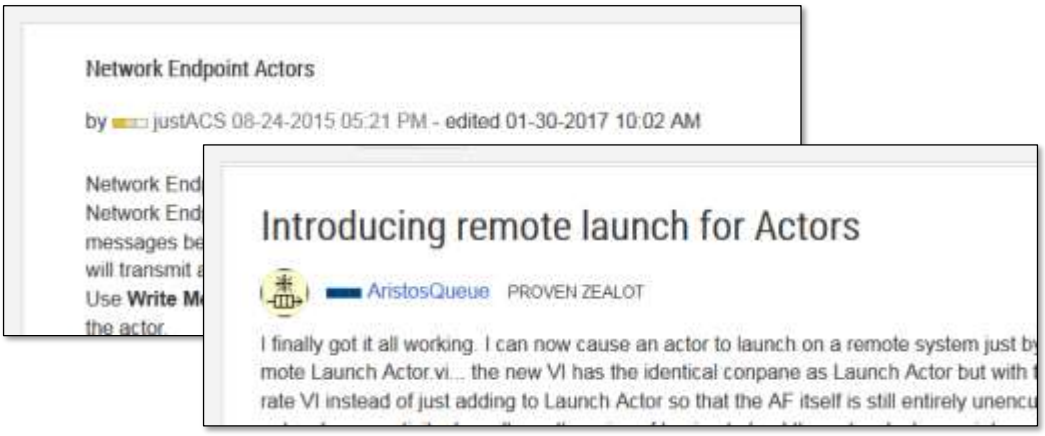
Sidebar: Distributing Actors for Reuse



NATIONAL INSTRUMENTS

The NI LabVIEW Tools Network maintains a VIPM repository where you can find the reuse actors used in this course (among other great downloads). Select Not Installed, and search for “actor” to get a list of the available actor packages.

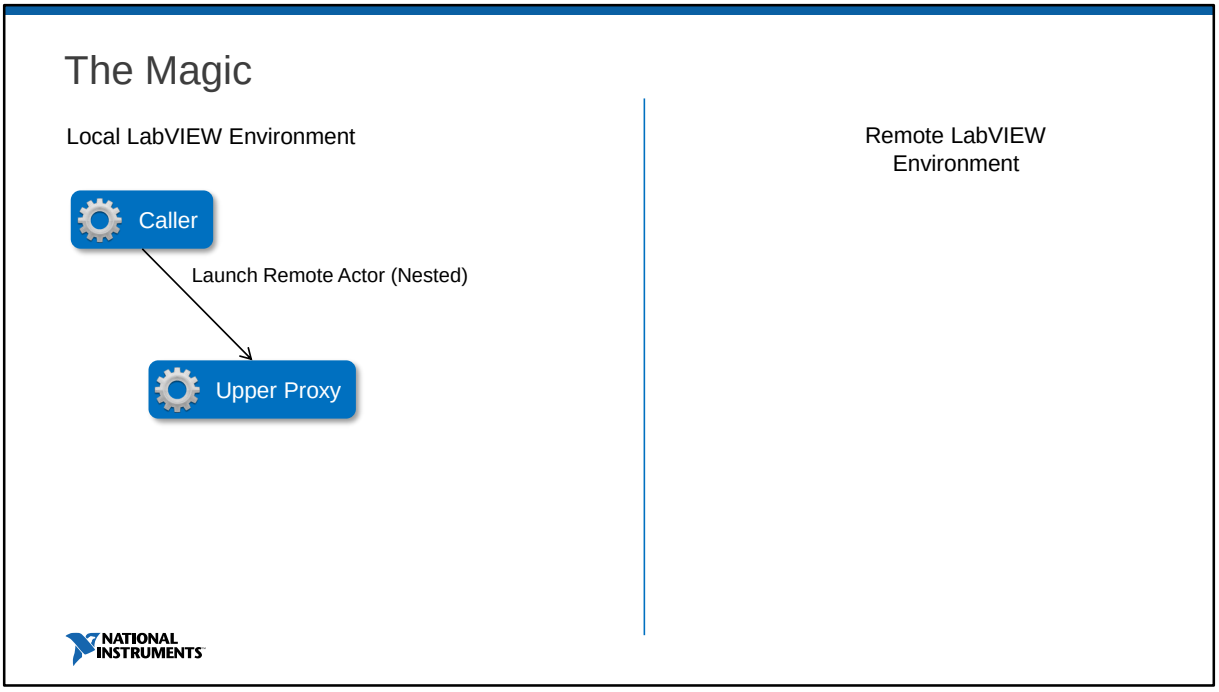
Actor Framework Forums



They can also be found on the forums, with notes about their use, but these versions are not updated.

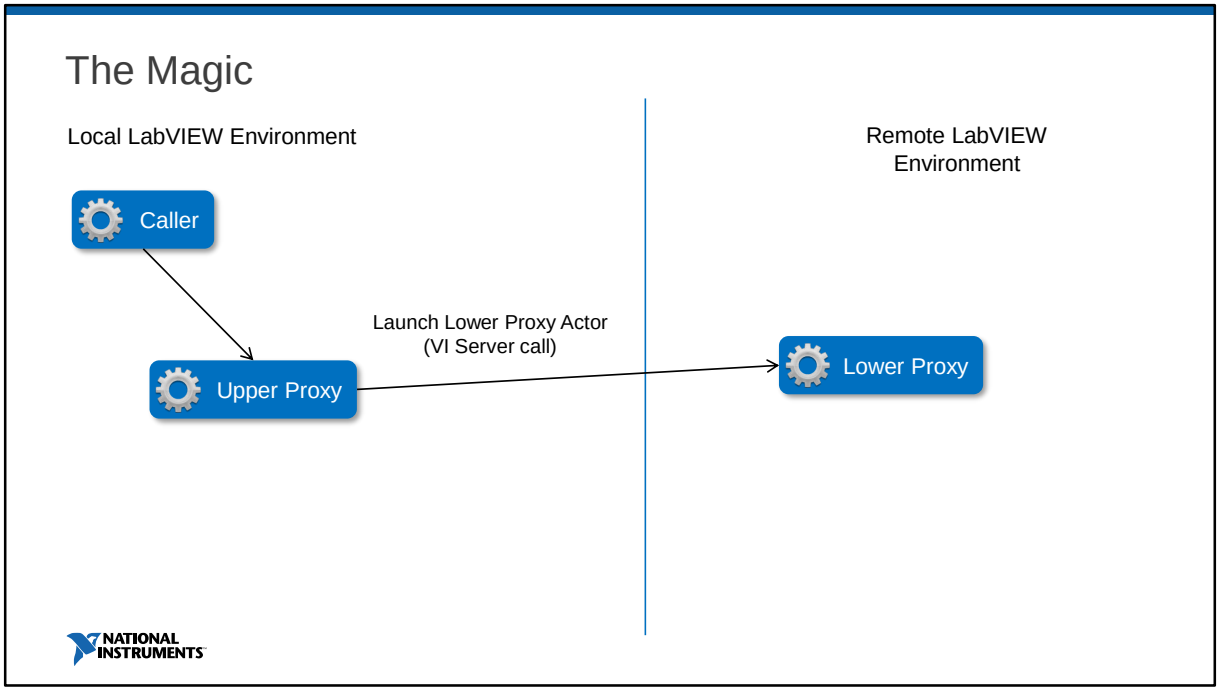
The easiest way to find them is to search ni.com for the document titles.

Appendix:
Launch Remote Actor - Under the Hood

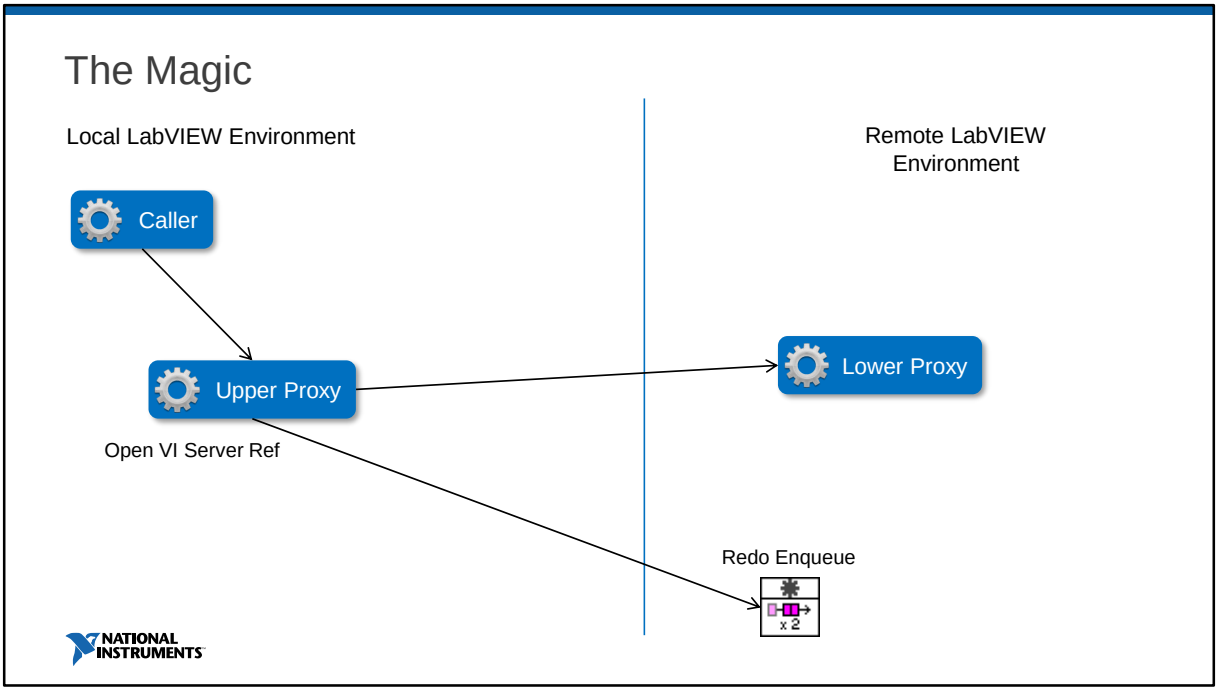


Launch Remote Actor relies on a set of VI Server calls to establish communication between the applications. These next few slides show th launch sequence.

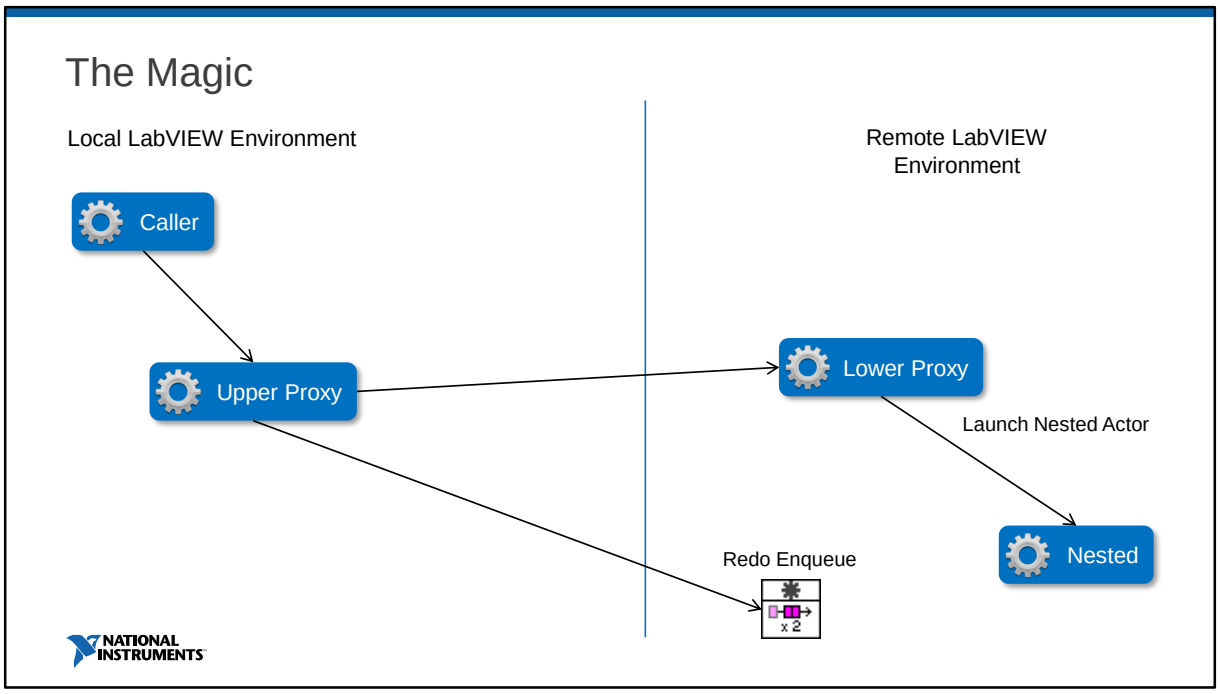
First, the local Caller invokes Launch Remote Actor, with an instance of the desired actor as an argument. Launch Remote Actor actually launches an instance of Upper Proxy Actor.



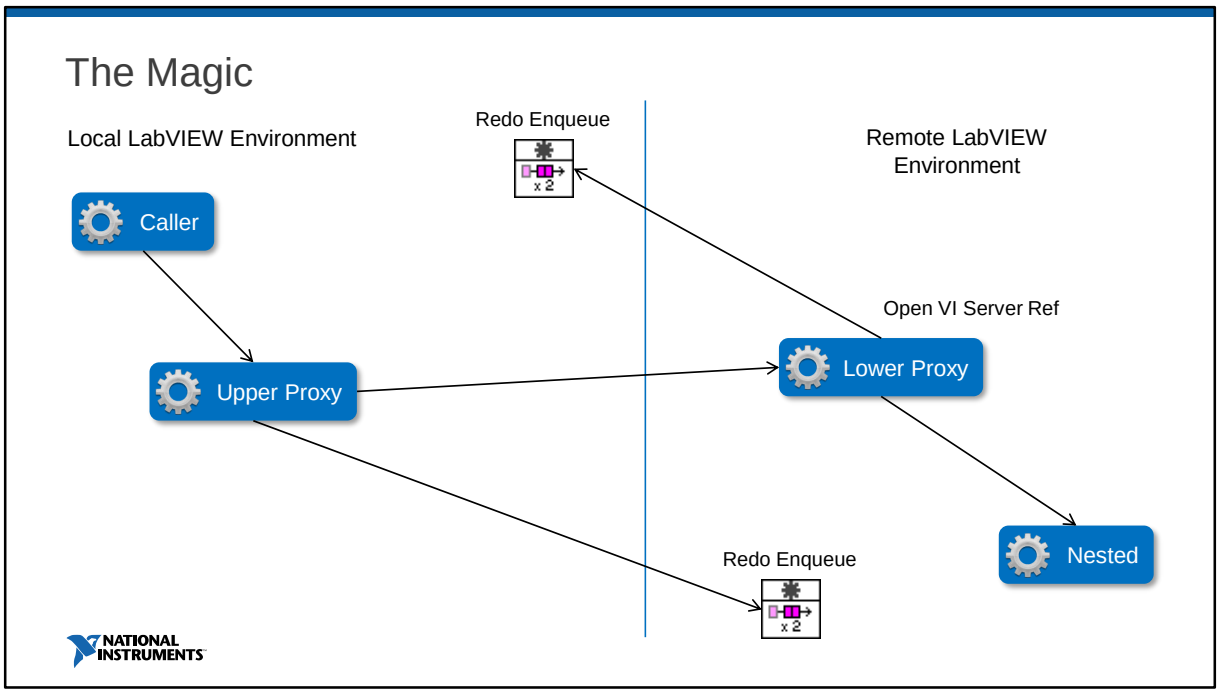
Upper Proxy Actor uses VI Server to open a reference to Launch Lower Proxy Actor, in the remote application instance. It remotely invokes this VI to launch the Lower Proxy Actor, and passes it the instance of the desired actor.



Upper Proxy also opens and holds a reference to a VI called Redo Enqueue.vi. This instance resides in the Lower Proxy's application instance

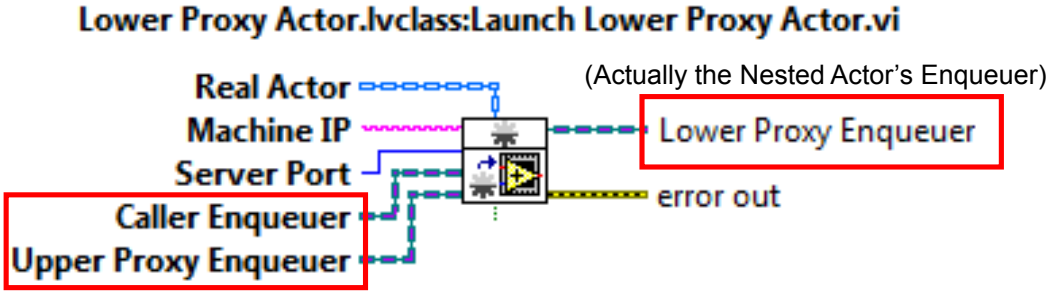


Lower Proxy Actor launches the desired actor as its nested actor…



… and opens a reference to an instance of Redo Enqueue in the Upper Proxy's application instance.

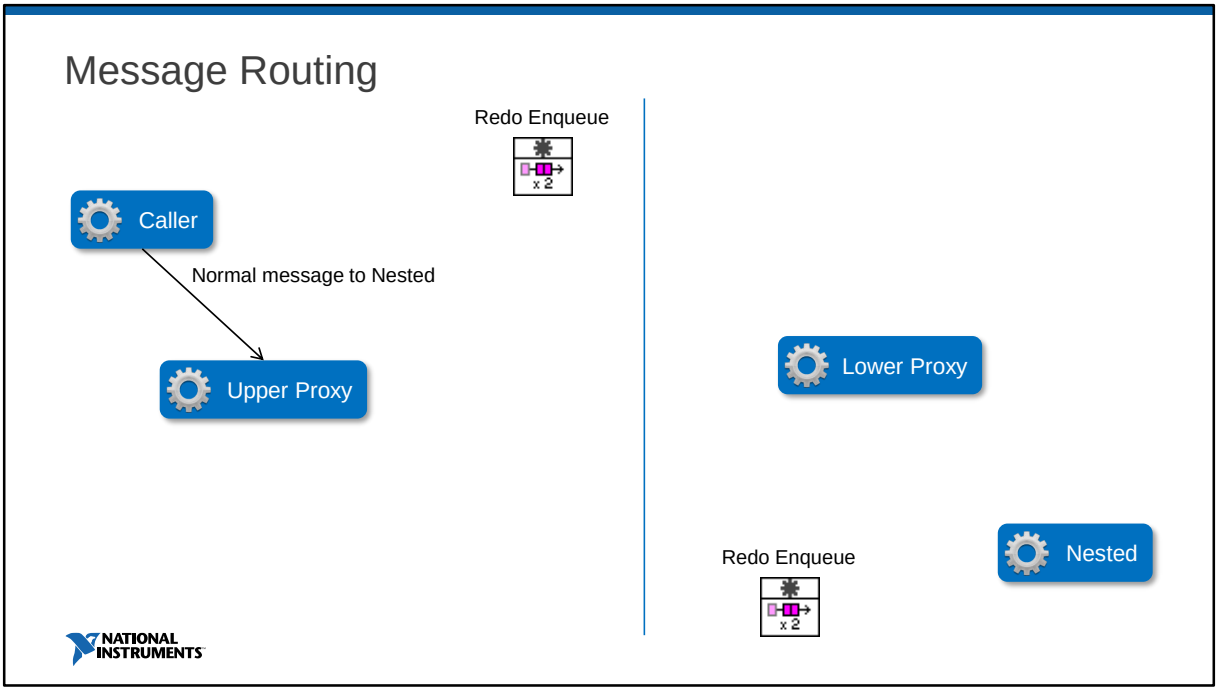
Launch Lower Proxy Actor



NATIONAL INSTRUMENTS

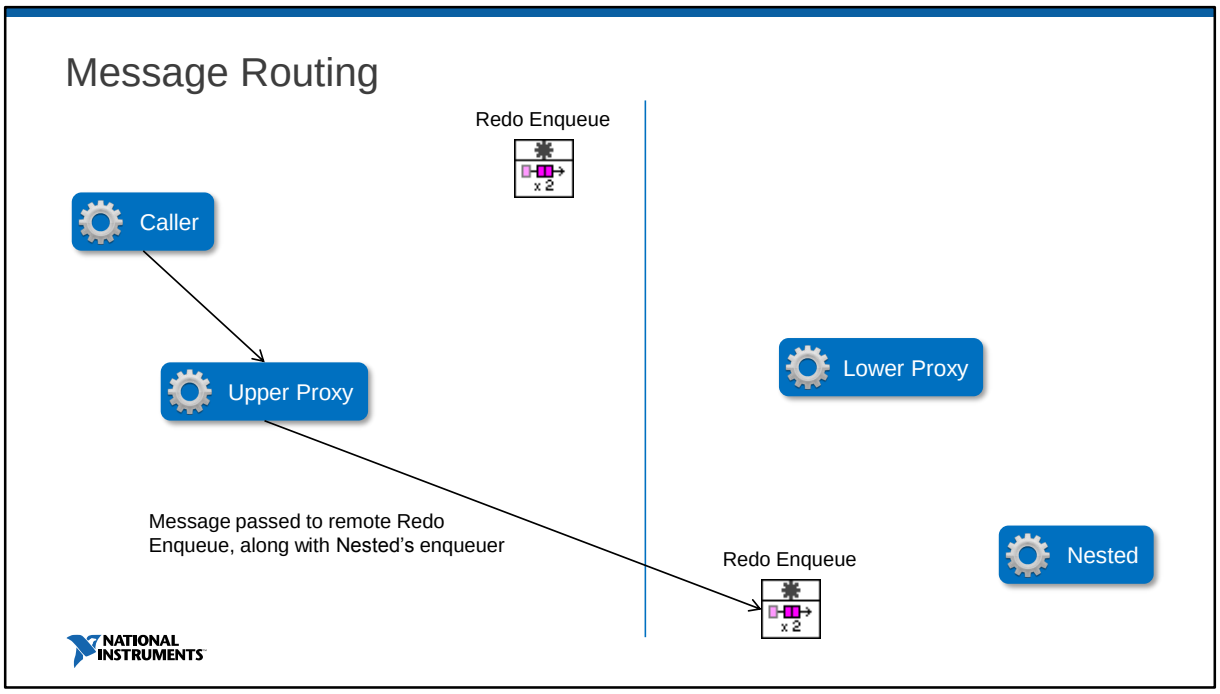
Upper Proxy Actor launches Lower Proxy Actor by calling Launch Lower Proxy Actor; this is done through a VI Server call. Upper Proxy Actor passes its own and its caller's enqueuers to the Lower Proxy, and the Lower Proxy returns the enqueuer of the actual nested actor (the output is mislabeled).

The enqueuers are useless outside their original application instances, of course. But their data is intact. Through VI Server calls, they can be passed back as arguments, and will work just fine. We use this fact to route messages.

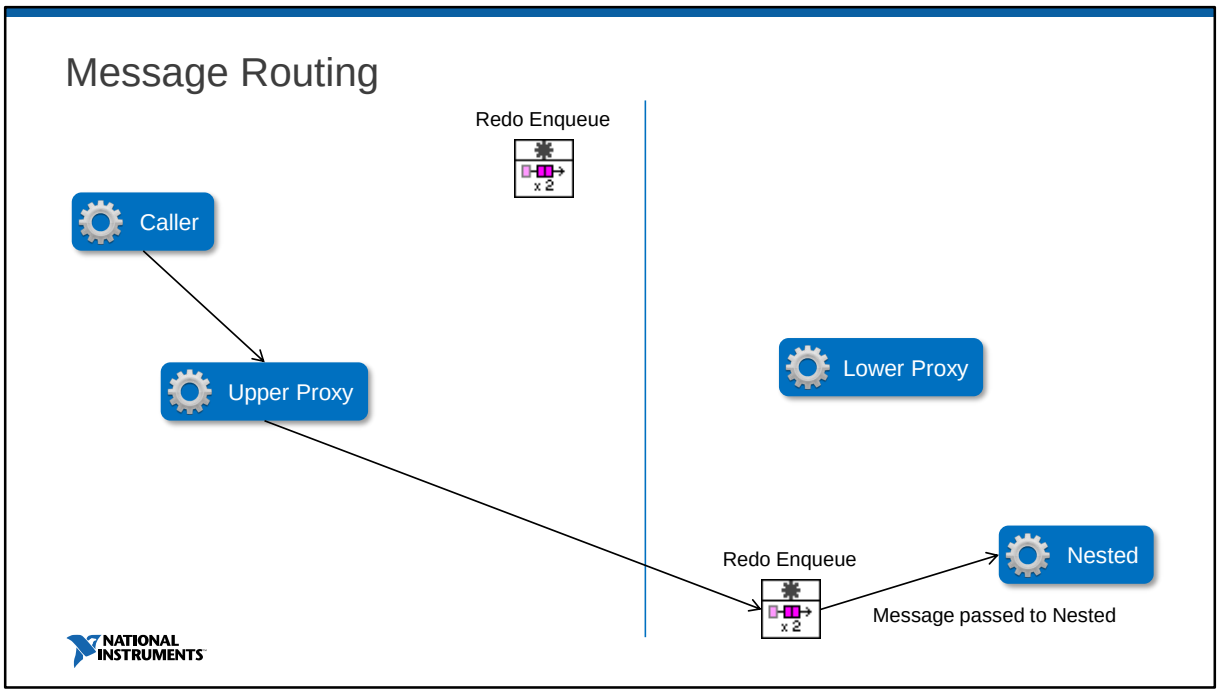


These next slides show how messages from Caller to Nested.

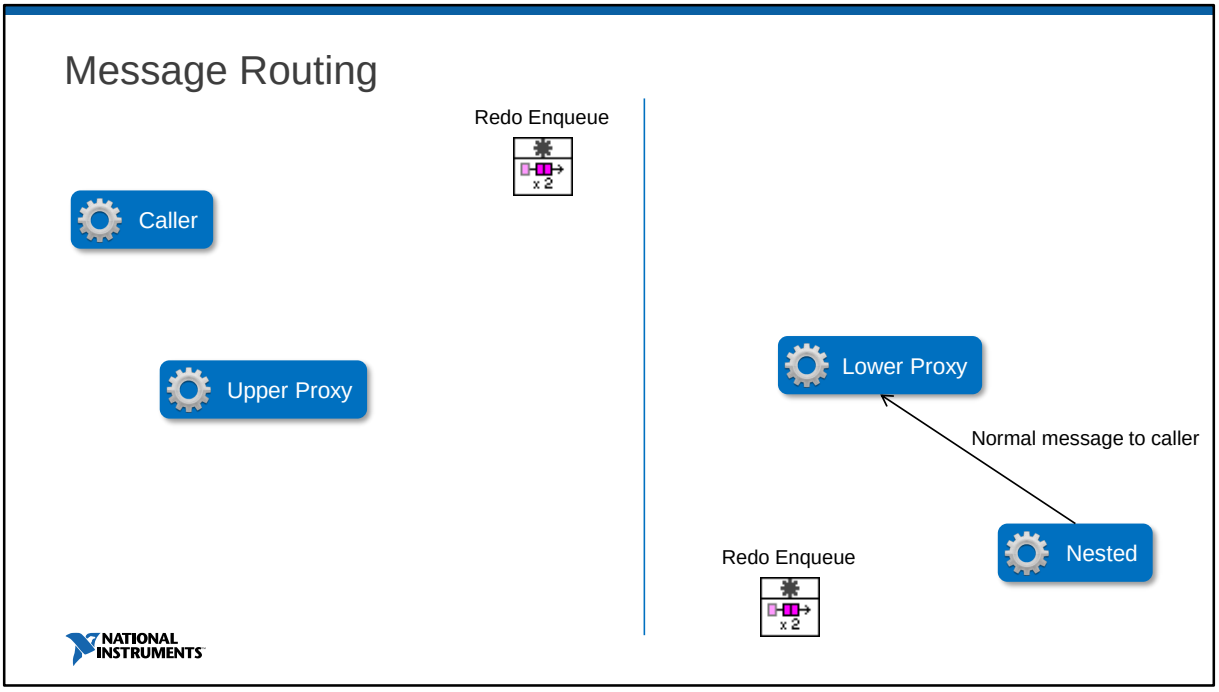
First, the Caller sends a normal message to the enqueuer it has for Nested, which is actually Upper Proxy's enqueuer.



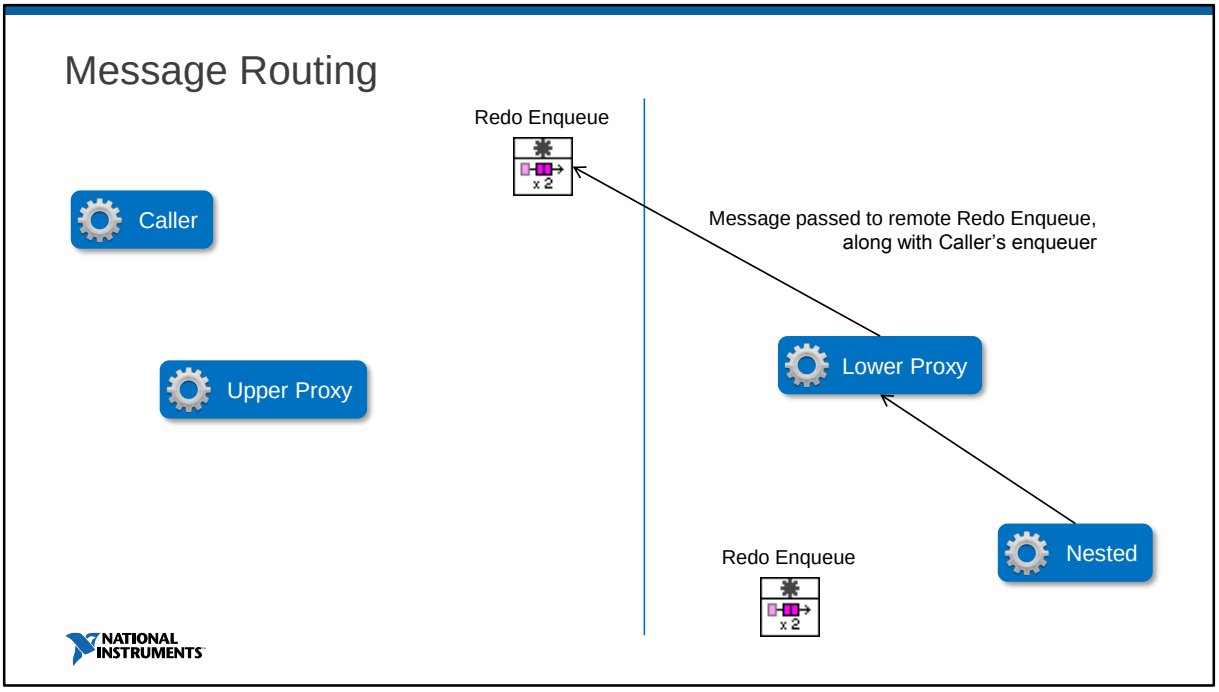
Upper Proxy invokes Redo Enqueue on the remote machine, passing the message and Nested's enqueuer as arguments. Recall that Upper Proxy got this as output from the call to Launch Lower Proxy Actor.



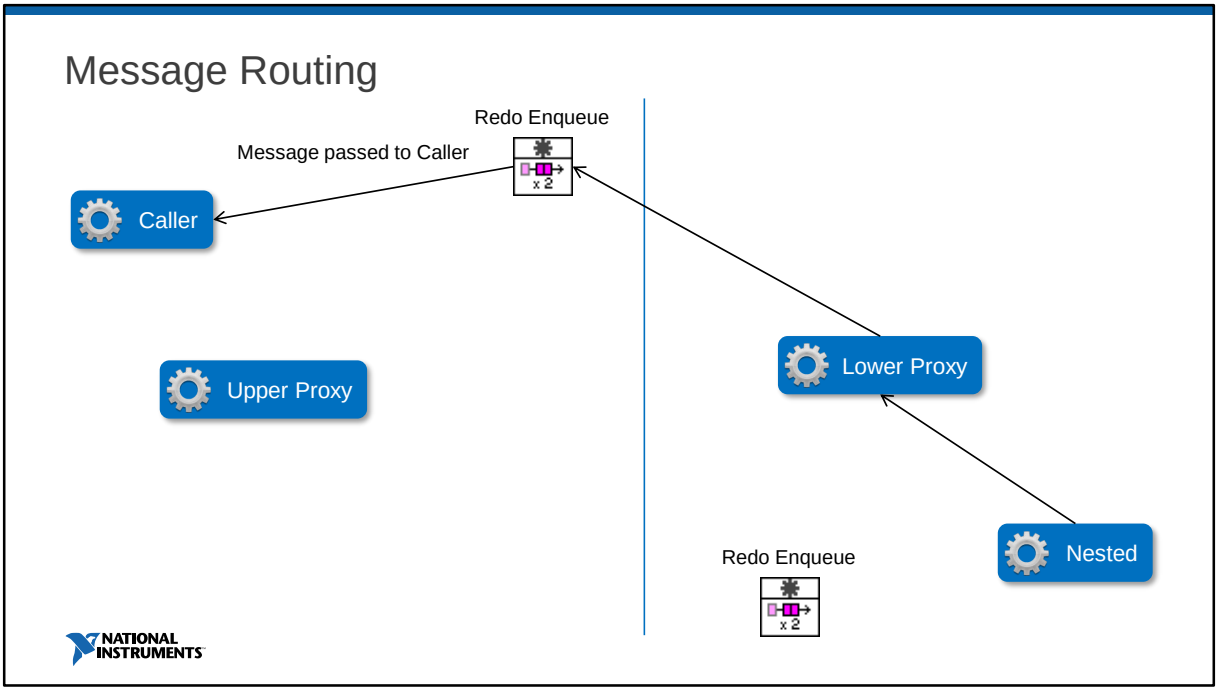
Redo Enqueue passes the message to Nested, with a normal Enqueue call.



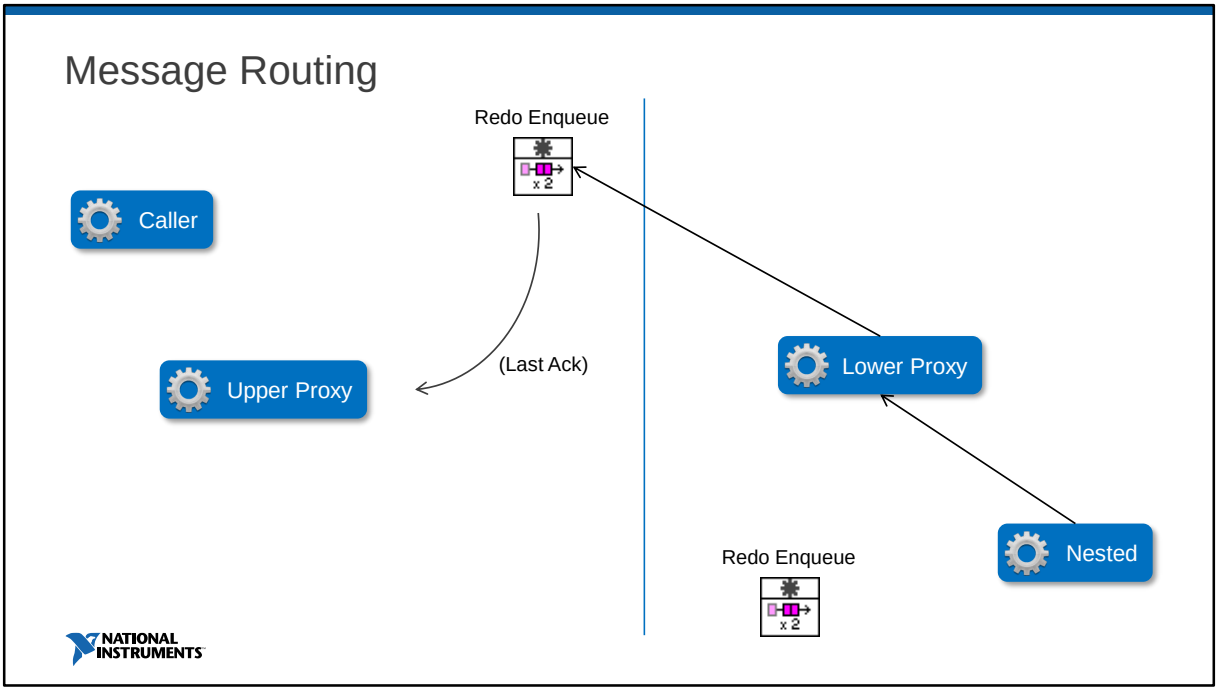
Messages from Nested to Caller move in a similar fashion. First, Nested sends messages to its caller, Lower Proxy Actor, as normal.



Lower Proxy invokes Redo Enqueue on the remote machine, passing the message and Caller's enqueuer as arguments. Caller passed its enqueuer to Lower Proxy Actor when it invoked Launch Lower Proxy Actor.vi.



Redo Enqueue passes the message to Caller, with a normal Enqueue call.



To ensure proper function, Lower Proxy's Last Ack is routed to Upper Proxy, and not to Caller.