REMOTING

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Table of Contents

1. Introduction
2. Akka.Remote Capabilities & Configuration
3. The AkkaTime Example
4. Remote Actor Communication and Deployment in Practice
5. Securing Akka.Remote Applications
6. Distributed Systems with Akka.Remote
7. Working With The Realities Of The Network

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Petabridge Akka.NET Remoting

In this Petabridge training course you will learn how to use **Akka.Remote**, Akka.NET's remoting library, to build distributed applications that are responsive, resilient to failure, and easy to program!

Concepts You'll Learn

The goal of *Akka.NET Remoting* is to teach you how Akka.Remote works and how to use its capabilities to effectively build both peer-to-peer and client-server applications.

Thus we've structured the course to do the following:

- **Teach you how Akka.Remote transports work and are configured** - what types of network transports are available? How do I configure them? What else is involved?
- **Show you how to communicate with and deploy remote actors** - how do I establish communication with a remote actor system? What do you mean by "deploy" an actor remotely? What's possible in this?
- **Give you a framework for when you should deploy vs. message remotely** - when should I use remote deployment? What about remote messaging? How do I track remote communication between actors? (*hint*: the answer is shockingly easy)
- **Discover strategies for securing your Akka.Remote applications** - what are my options with respect to security? What are the best practices?
- **Learn the basics of building distributed systems** - where should my state live? What sort of consistency model can I live with? How do I deal with failures?
- **Deal powerfully with the realities of the network: latency, disconnects, and dead processes** - how do I know if a remote actor died? How do I deal with latency? How do I recover from a disconnect? How do I know if my message was received?

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

After completing this course you should have enough familiarity with Akka.Remote to be able to use it inside your own applications, complete with strategies for anticipating and dealing powerfully with the realities of imperfect networks.

You'll be exposed to Akka core, Akka.Remote, and Akka.Persistence.

> If you want to learn how to combine traditional Akka.NET design patterns with Akka.Remote or how to use Akka.Cluster for large-scale server-side applications, take our Akka.NET design patterns course and our Akka.Cluster training course. These will both help you really understand how distributed systems built on top of Akka.NET actually work and will help eliminate sources of DevOps issues for you in the future.

How This Course Works

Unlike Akka.NET Bootcamp, this course is not intended to be self-directed learning. This is meant to be done in real-time with an instructor in a small group.

The course works in the following way:

1. There will be a period of lecture plus presentation by the instructor in which you'll be introduced to new concepts, during which you can ask questions about the material;
2. Afterwards you'll do an interactive design exercise with the instructor, where you'll apply the concepts learned during the previous lecture;
3. Following a quick review of the exercise, the results of that exercise will be applied to a specific application the instructor will be iteratively building throughout the course;
4. Rinse and repeat until all lectures have been completed.

Once the live portion of the course is complete, you'll receive all of the code samples and a written handout that covers everything you learned in the course so you can review it all later.

Make Sure You Ask Questions!

You won't get the full value out of this course if you don't ask the instructor questions!

The instructor has budgeted 2.5-3 hours to go through all of the official Petabridge

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

training material, but the training course is meant to be 4-4.5 hours long.

So make sure you ask questions along the way! It's your time - make sure you take full advantage of it!

What You Need to Do Before the Training

Make sure you have the following available during the training:

1. **Paper and pen** - we'll be doing some drawing and modeling exercises on paper;
2. **Visual Studio 2012/2013 and SQL Server Express 2012** - so you can run the downloadable samples after the training and make changes to the code; and
3. **A good pair of headphones and a microphone** so your questions can be clearly heard by the other attendees and the instructor. If you don't have access to a microphone you will be able to type in your questions, but *we strongly recommend* having a good microphone.

Table of Contents

1. **Akka.Remote Capabilities and Configuration**
   i. Transports, Sockets, and Akka.NET `Address`es
   ii. Remote Communication between `ActorSystem`s
   iii. Remote Deployment between `ActorSystem`s
2. **The AkkaTime Example**
3. **Remote Actor Communication and Deployment in Practice**
4. **Securing Akka.Remote Applications**
5. **Distributed Systems with Akka.Remote**
   i. Peer-to-Peer Architectures
   ii. Client-Server Architectures
6. **The Realities of the Network**
   i. Working with Latency
   ii. Working with Network Failures
   iii. Working with Dead / Hung Processes

Copyrighted Material

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

All of the material used in this training is copyrighted by Petabridge LLC.

The handouts and any materials distributed afterwards are intended for your personal use, not to be shared or redistributed.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Akka.Remote Capabilities and Configuration

Akka uses a the "Home Depot" model towards application design - in addition to the base Akka NuGet package there are dozens of additional modules you can add to your project that take the capabilities of Akka and extend it to do new things!

Akka.Remote is the most powerful of all of these additional packages, as it is what brings the capability to build an `ActorSystem` across multiple processes over a computer network.

In this section we're going to explore the capabilities and configuration details that allow you to use Akka.Remote inside your Akka.NET applications!

A Trivial Akka.Remote Example

We're going to begin our exploration of Akka.Remote by taking a look at a trivial Akka.Remote sample directly from the core Akka.NET project.

This sample is a basic command-line chat room, and it's been used to demonstrate the lowest-level Akka.Remote capabilities for years.

Here's the three core pieces of code that make this sample work:

Chat Messages (shared between Client and Server)

```
namespace ChatMessages
{
    public class ConnectRequest
    {
        public string Username { get; set; }
    }

    public class ConnectResponse
    {
        public string Message { get; set; }
    }

    public class NickRequest
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
    {
        public string OldUsername { get; set; }
        public string NewUsername { get; set; }
    }

    public class NickResponse
    {
        public string OldUsername { get; set; }
        public string NewUsername { get; set; }
    }

    public class SayRequest
    {
        public string Username { get; set; }
        public string Text { get; set; }
    }

    public class SayResponse
    {
        public string Username { get; set; }
        public string Text { get; set; }
    }

    public class ChannelsRequest
    {
    }

    public class ChannelsResponse
    {
        public IActorRef[] channels { get; set; }
    }

    public class Disconnect
    {
    }
}
```

The `ChatMessages` project gets shared between both the `ChatClient` and the `ChatServer`, because otherwise neither would be able to deserialize the other's messages.

Chat Client

```
namespace ChatClient
{
    class Program
    {
        static void Main(string[] args)
        {
            var config = ConfigurationFactory.ParseString(@"
akka {
    actor {
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }
    remote {
        helios.tcp {
            transport-class = ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote
            applied-adapters = []
            transport-protocol = tcp
            port = 0
            hostname = localhost
        }
    }
}
");

            using (var system = ActorSystem.Create("MyClient", config))
            {
                var chatClient = system.ActorOf(Props.Create<ChatClientActor>());
                system.ActorSelection("akka.tcp://MyServer@localhost:8081/user/ChatServer");
                chatClient.Tell(new ConnectRequest()
                {
                    Username = "Roggan",
                });

                while (true)
                {
                    var input = Console.ReadLine();
                    if (input.StartsWith("/"))
                    {
                        var parts = input.Split(' ');
                        var cmd = parts[0].ToLowerInvariant();
                        var rest = string.Join(" ",parts.Skip(1));

                        if (cmd == "/nick")
                        {
                            chatClient.Tell(new NickRequest
                            {
                                NewUsername = rest
                            });
                        }
                    }
                    else
                    {
                        chatClient.Tell(new SayRequest()
                        {
                            Text = input,
                        });
                    }
                }
            }
        }
    }

    class ChatClientActor : TypedActor,
        IHandle<ConnectRequest>,
        IHandle<ConnectResponse>,
        IHandle<NickRequest>,
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        IHandle<NickResponse>,
        IHandle<SayRequest>,
        IHandle<SayResponse>, ILogReceive
    {
        private string _nick = "Roggan";
        private readonly ActorSelection _server =
            Context.ActorSelection("akka.tcp://MyServer@localhost:8081/user/ChatServer");

        public void Handle(ConnectResponse message)
        {
            Console.WriteLine("Connected!");
            Console.WriteLine(message.Message);
        }

        public void Handle(NickRequest message)
        {
            message.OldUsername = this._nick;
            Console.WriteLine("Changing nick to {0}", message.NewUsername);
            this._nick = message.NewUsername;
            _server.Tell(message);
        }

        public void Handle(NickResponse message)
        {
            Console.WriteLine("{0} is now known as {1}", message.OldUsername, message.NewUse
        }

        public void Handle(SayResponse message)
        {
            Console.WriteLine("{0}: {1}", message.Username, message.Text);
        }

        public void Handle(ConnectRequest message)
        {
            Console.WriteLine("Connecting....");
            _server.Tell(message);
        }

        public void Handle(SayRequest message)
        {
            message.Username = this._nick;
            _server.Tell(message);
        }
    }
}
```

The `ChatClient` is simple - it connects to the `ChatServer` and just reads input from the commandline in a loop, sending messages to and printing responses from the server.

Chat Server

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
namespace ChatServer
{
    class Program
    {
        static void Main(string[] args)
        {
            var config = ConfigurationFactory.ParseString(@"
akka {
    actor {
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }
    remote {
        helios.tcp {
            transport-class = ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote
            applied-adapters = []
            transport-protocol = tcp
            port = 8081
            hostname = localhost
        }
    }
}
");

            using (var system = ActorSystem.Create("MyServer", config))
            {
                system.ActorOf<ChatServerActor>("ChatServer");

                Console.ReadLine();
            }
        }
    }

    class ChatServerActor : TypedActor ,
        IHandle<SayRequest>,
        IHandle<ConnectRequest>,
        IHandle<NickRequest>,
        IHandle<Disconnect>,
        IHandle<ChannelsRequest>,
        ILogReceive

    {
        private readonly HashSet<IActorRef> _clients = new HashSet<IActorRef>();

        public void Handle(SayRequest message)
        {
          //  Console.WriteLine("User {0} said {1}",message.Username , message.Text);
            var response = new SayResponse
            {
                Username = message.Username,
                Text = message.Text,
            };
            foreach (var client in _clients) client.Tell(response, Self);
        }

        public void Handle(ConnectRequest message)
```

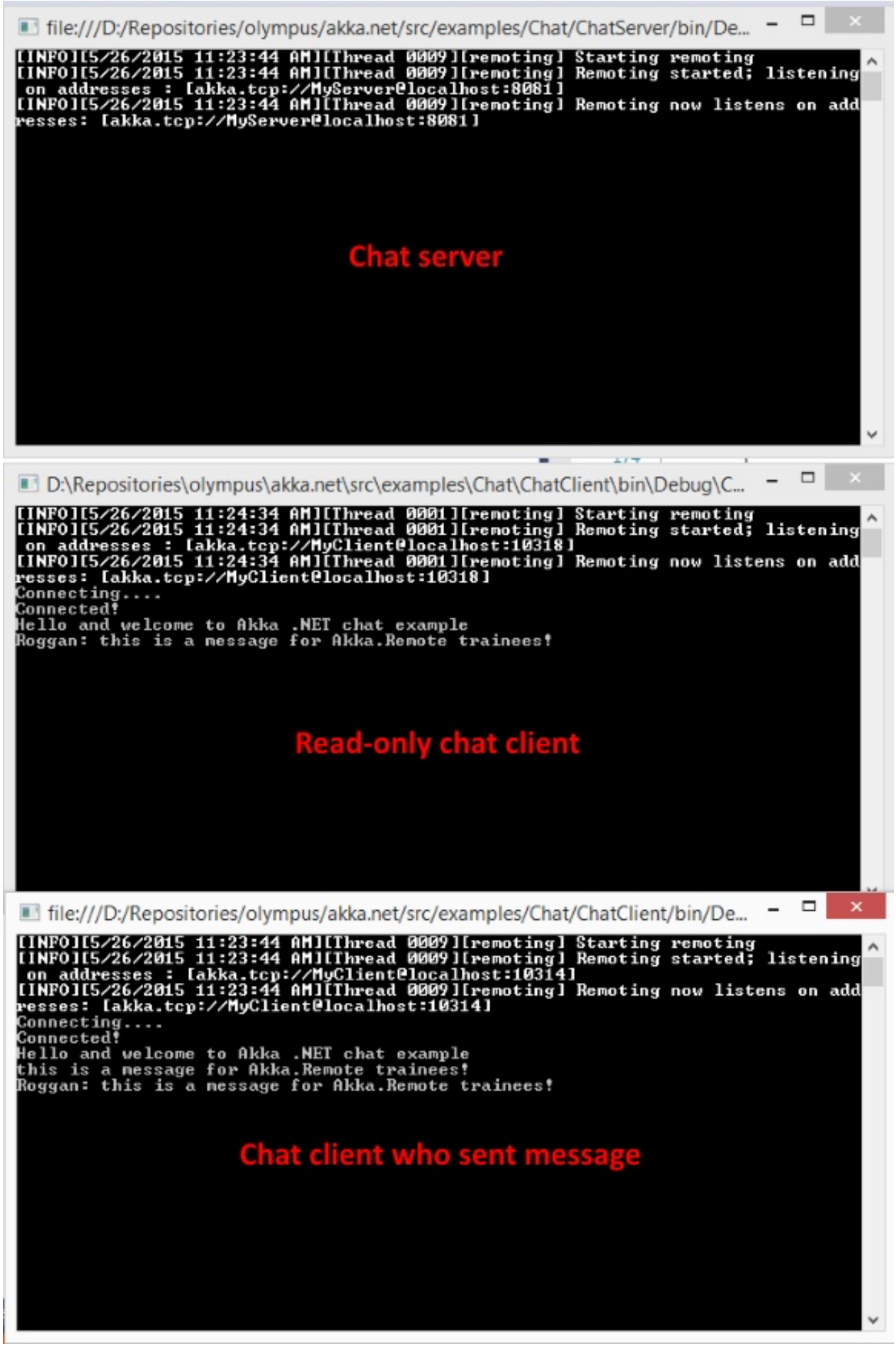

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        {
         //   Console.WriteLine("User {0} has connected", message.Username);
            _clients.Add(this.Sender);
            Sender.Tell(new ConnectResponse
            {
                Message = "Hello and welcome to Akka .NET chat example",
            }, Self);
        }

        public void Handle(NickRequest message)
        {
            var response = new NickResponse
            {
                OldUsername = message.OldUsername,
                NewUsername = message.NewUsername,
            };

            foreach (var client in _clients) client.Tell(response, Self);
        }

        public void Handle(Disconnect message)
        {

        }

        public void Handle(ChannelsRequest message)
        {

        }
    }
}
```

The `ChatServer` in this case simply echos the messages it receives from the `ChatClient` back to it, but with a *twist*! It'll broadcast the chat messages it receives back to **all** connected `ChatClients` simultaneously.

Here's a screenshot of what it looks like in action, along with two connected `ChatClient` instances.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

So what's interesting about this? The place we should look first is the HOCON configuration we're using to start the `ActorSystem` for both the `ChatClient` and `ChatServer`.

Server

```
akka {
    actor {
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }
    remote {
        helios.tcp {
            transport-class = ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote
            applied-adapters = []
            transport-protocol = tcp
            port = 8081
            hostname = localhost
        }
    }
}
```

Client

```
akka {
    actor {
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }
    remote {
        helios.tcp {
            transport-class = ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote
            applied-adapters = []
            transport-protocol = tcp
            port = 0
            hostname = localhost
        }
    }
}
```

We did two things in this configuration that are interesting:

1. We swapped out the normal `akka.actor.provider` for the `Akka.Remote.RemoteActorRefProvider` in both projects; and
2. We enabled the `helios.tcp` transport and configured it to bind on *different addresses* in each project.

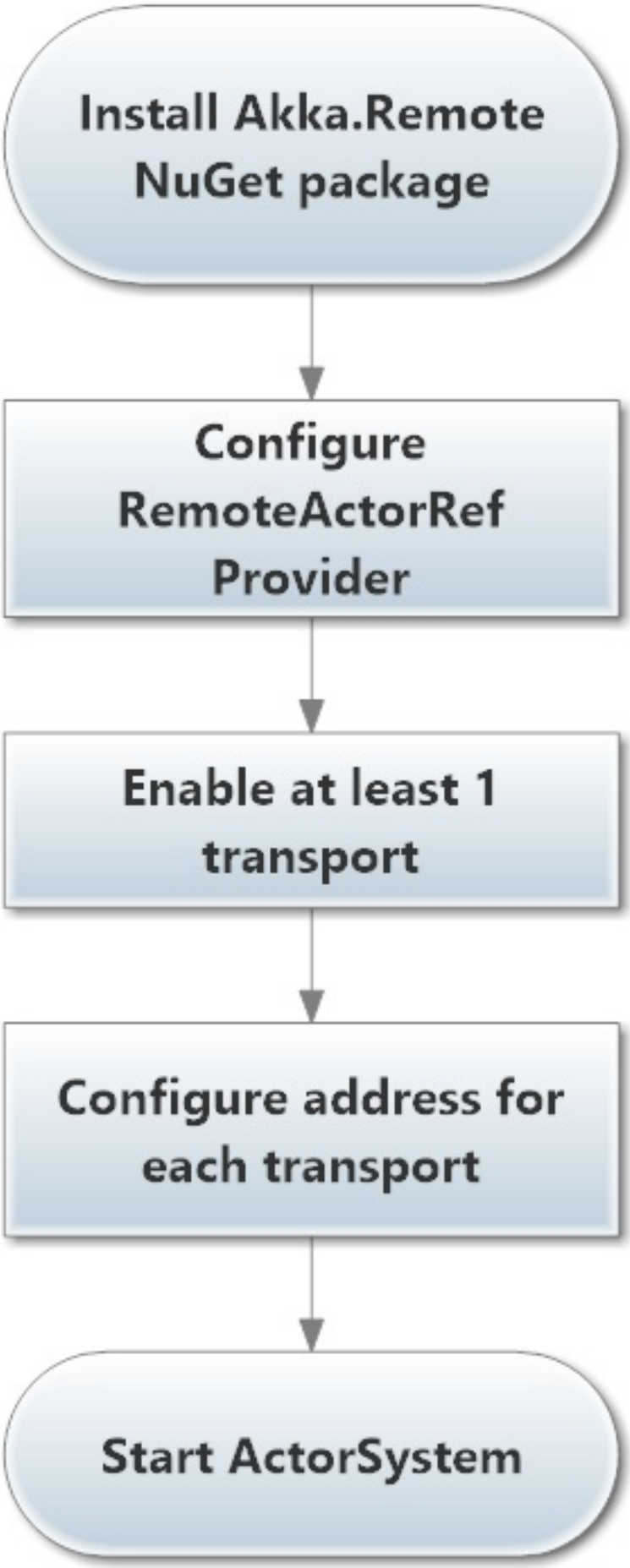
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

What's that all about?

How to Turn on Akka.Remote

If you've already played with Akka.Remote a bit, then you might already be familiar with these steps. But in case you've not used Akka.Remote much before, we're going to make the steps for enabling Akka.Remote explicit.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

The first step is to just install the Akka.Remote NuGet package, which you can do via the following command in the Package Manager Console in Visual Studio:

```
PM> Install-Package Akka.Remote
```

The next step is to configure your `ActorSystem` to use the `RemoteActorRefProvider`, which we do here in configuraiton:

```
akka {
    actor {
        #only use single quotes if loading config from file
        provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
    }
}
```

This is what turns on all of the plumbing needed to run remoting - we'll get onto the details of that a little bit later.

> **NOTE**: It's important to note that for certain types of settings in Akka.NET, such as the `akka.actor.provider` setting, you have to use a Fully Qualified Type Name (FQN) in configuration. Read more about FQNs here.

Lastly, we have to enable a transport and bind it to an address. This is what we're doing here:

```
akka{
    remote {
        helios.tcp {
            transport-class = ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote
            applied-adapters = []
            transport-protocol = tcp
            port = 0
            hostname = localhost
        }
    }
}
```

Once that's done, we can start our `ActorSystem` and begin communicating remotely with other processes with Akka.Remote!

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Akka.Remote Transports, Endpoints, and Associations

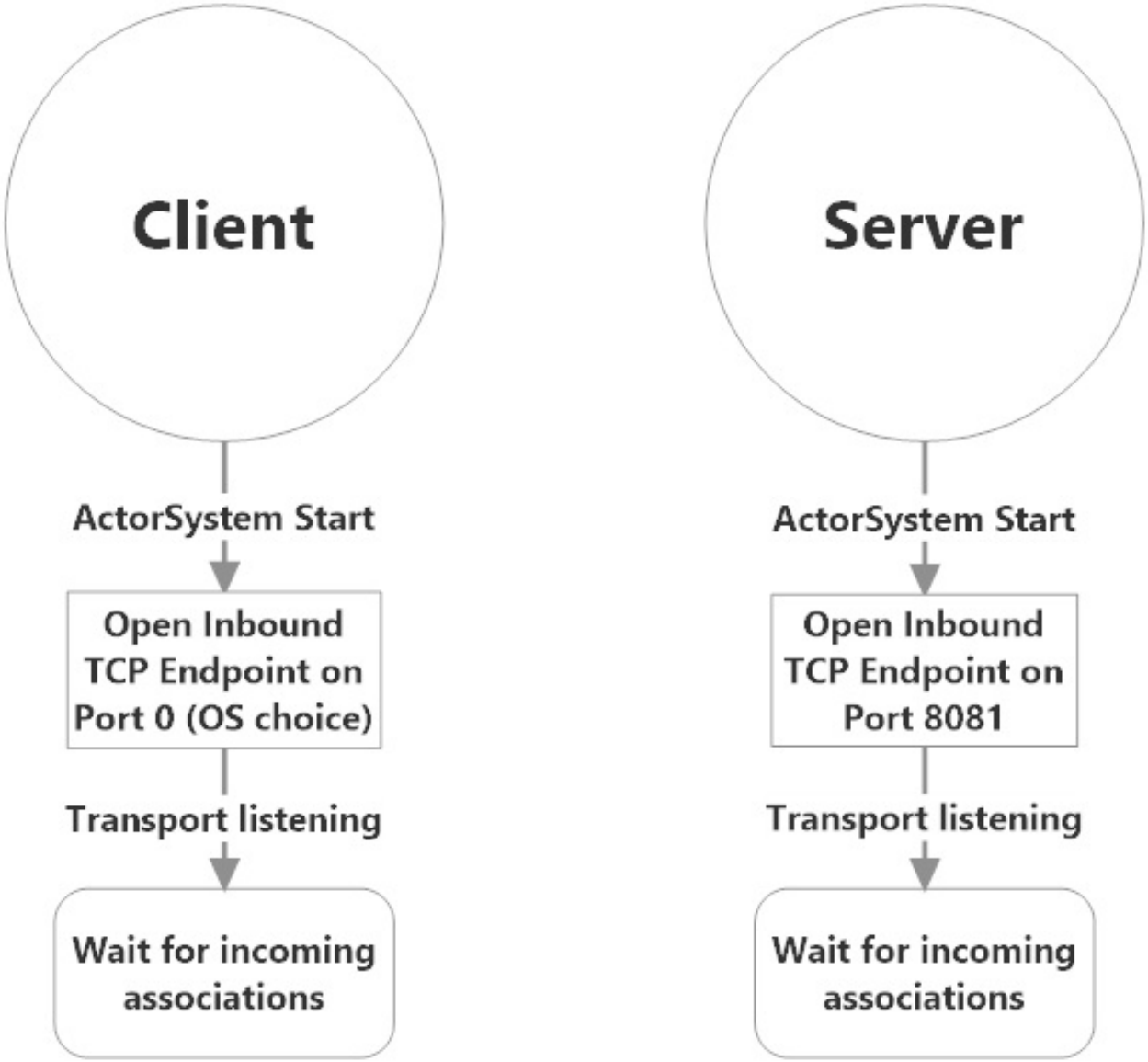
So before we dive into the guts of how Akka.Remote works, we need to share some of the terms it uses to describe the network.

- **Transport** - a "transport" refers to an actual network transport, such as TCP or UDP. By default Akka.Remote uses a Helios TCP transport, but you could write your own transport and use that instead of you wish.
- **Address** - this refers to an IP address and port combination, just like any other IP-enabled protocol. You can also use a hostname instead of an IP address, but the hostname must be resolved to an IP address first.
- **Endpoint** - an "endpoint" is a specific address binding for a transport. If I open a TCP transport at `localhost:8080` then I've created an *endpoint* for that transport at that address.
- **Association** - an "association" is a connection between two endpoints, each belonging to a different `ActorSystem`. Must have a valid *outbound* endpoint and a valid *inbound* endpoint in order to create the association.

These terms form the basis for all remote interaction between `ActorSystem` instances, so they're critically important to learn and distinguish.

Given all of that, let's consider what the initial state of two two remote `ActorSystem`s from the previous trivial example look like when they first boot up.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Both `ActorSystem` instances start, open their transports and bind them to the configured addresses (which creates an *inbound* endpoint for each) and then waits for incoming association attempts from elsewhere.

The Magic of Port Zero

Before we move onto the other parts of the association process, it's worth talking about what we did on the `ChatClient` - *we bound the TCP transport to port zero.*

Why is that important? Port 0 tells the operating system to *pick a randomly available port* for you. You have no idea what the # is going to be, but upon startup the `ChatClient` process will be given a valid port to use for its inbound endpoint. And the reason why this information is important? Because the *inbound endpoint **is** the network identity of the* `ChatClient` `ActorSystem`.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

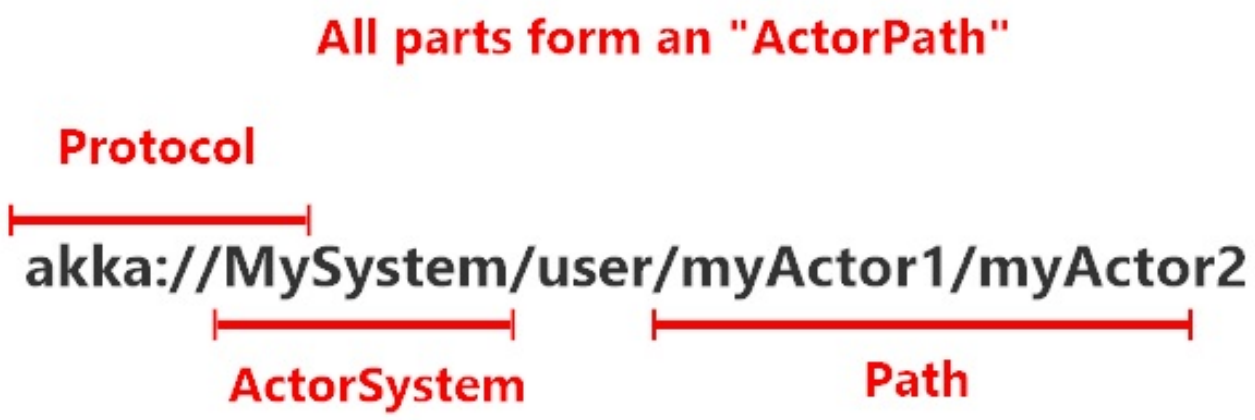
By binding to port 0, we can start multiple `ChatClient` instances in parallel and all have them connect to the server without any configuration changes.

Connecting to Remote `ActorSystem`s

In order to form an association with a remote `ActorSystem`, we have to have an `Address` for that `ActorSystem`.

All local Akka.NET actors have an `Address` too, as part of their `ActorPath`.

A local `ActorPath` look like this:



A remote `ActorPath` looks like this:

Each `ActorPath` consists of four parts:

1. **Protocol** - this defines the protocol used to communicate with this actor. Default local protocol is in-memory message passing.
2. **ActorSystem** - the name of the `ActorSystem` to which this actor belongs.
3. **Address** - refers to the inbound endpoint you can use to communicate with this actor via the protocol. There's a default address for local-only actors and it always get committed from local `ActorPaths`.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

4. **Path** - refers to the path of this actor in the hierarchy.

When you want to connect to a remote `ActorSystem`, two important changes occur to the address:

1. **The protocol gets augmented with the protocol of the network transport** - so in this case, since we're using the Helios TCP transport the protocol for communicating with all remote actors in our `ActorSystem` changes from `akka://` to `akka.tcp://`. When you deploy an actor remotely or send a message to a remote actor via `ActorSelection`, specifying this protocol is what tells your local `ActorSystem` how to deliver this message to the remote one!
2. **The address gets populated with the inbound endpoint on the transport** - `localhost:9001` in this case. This lets your local system know how to attempt to establish an *outbound endpoint* to the remote `ActorSystem`.

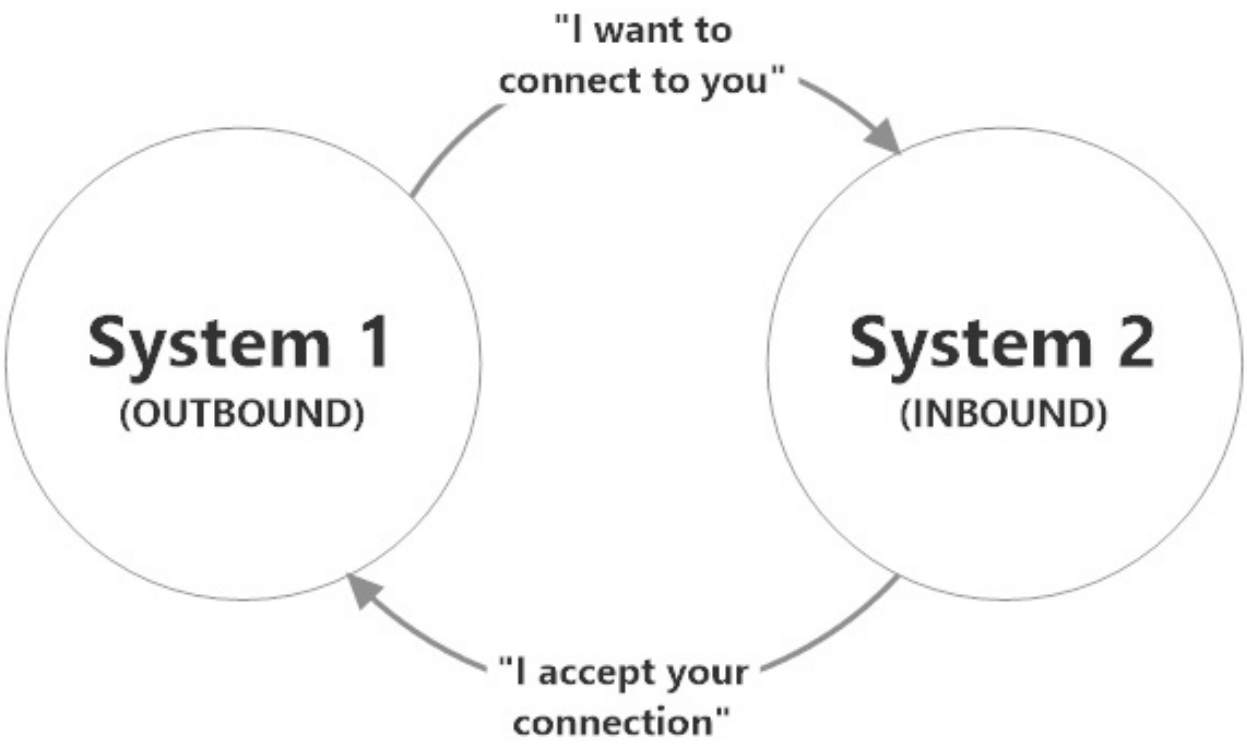
Let's take a closer look at the association process.

Inbound vs. Outbound Associations

We've mentioned the concept of *inbound* and *outbound* a few times now. What we do we mean by that?

This refers to the direction in which an association occurs. The `ActorSystem` initiating the association with another system is referred to as the *outbound* endpoint of the association. The `ActorSystem` receiving the association is referred to as the *inbound* endpoint of the association.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



An outbound association request is what it takes to start the association process.

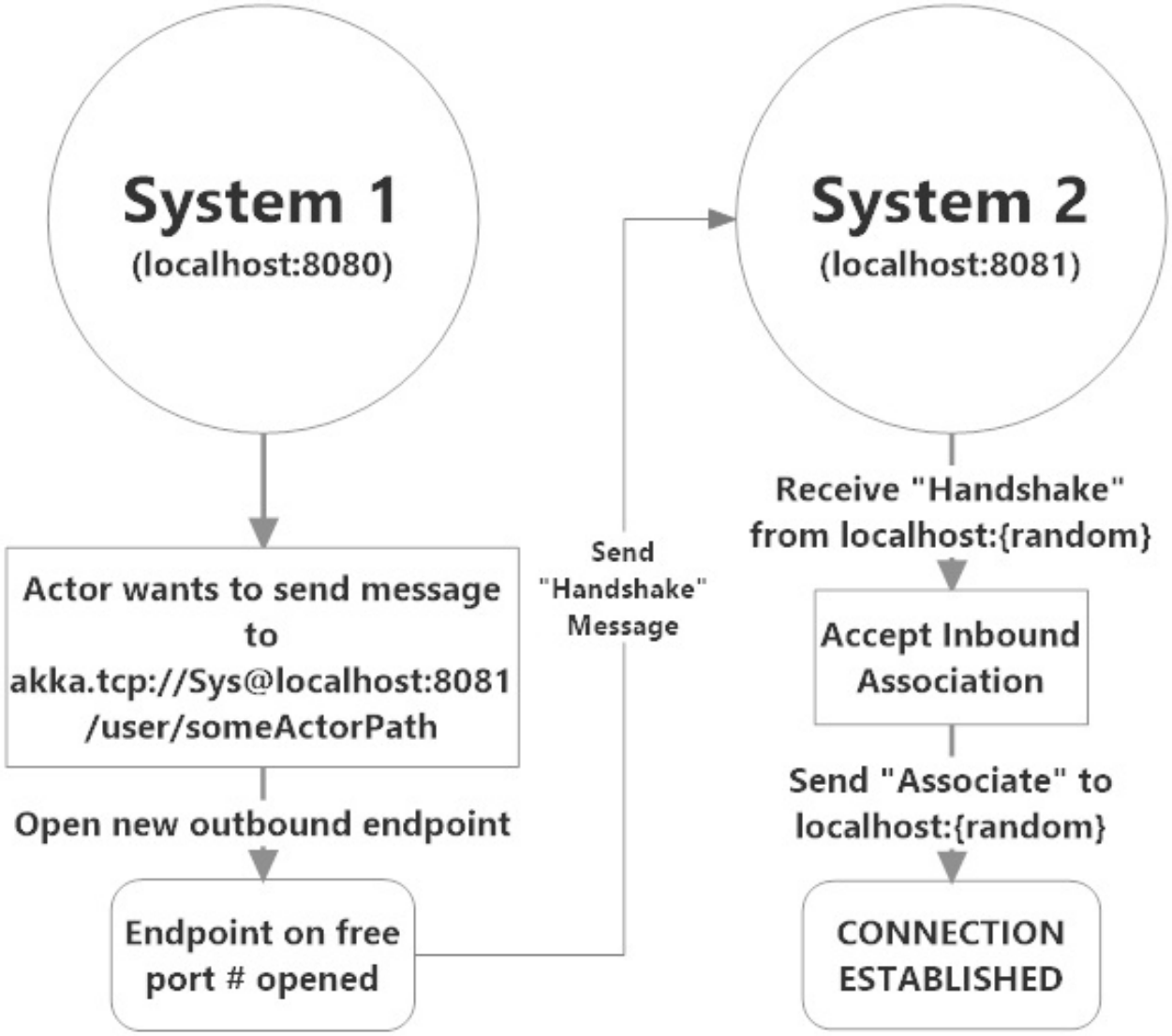
> **NOTE: Once this process is complete and both `ActorSystem`s are associated with each other, the direction of the association no longer matters - both systems can send messages to each other's actors freely.**

The Association Process

This information exposes some of the Akka.Remote internals to you, but it's important to know because without this information it's very difficult to troubleshoot association problems in production - *which you should anticipate as a product of imperfect networks*.

Just to repeat ourselves: **if you don't understand how Akka.Remote forms associations then when the network comes along and kills a process, times out, or does something else it will inevitably do you will not know how to debug your code and it will be your fault.** So pay attention to this part!

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



The association process begins when **System 1** has an actor who wants to send a message to an `ActorSelection` belonging to an actor who resides on **System 2**.

The `RemoteActorRefProvider` built into System 1, upon seeing the remote address in the `ActorSelection`, will check to see if a remote connection to System 2 is already open. Since there isn't one, it will open a new outbound endpoint using its TCP transport (which, internally, will create a new TCP socket on a new port - but that's beyond the scope of this course) and send an "handshake" message to System 2.

System 2 will receive the handshake, accept the inbound association, and reply with an "Associate" message which will complete the association process.

System 1 will then finally deliver the message contained in the `ActorSelection` to the appropriate actor on System 2.

That's how associations work in a nutshell!

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

> NOTE: If you'd like to learn more about how the Akka.Remote association process works, we recommend you check out ***Akka.NET Internals: How Akka.Remote Connections Work*** (22:35.)

Remote Communication: `ActorSelection` and `RemoteActorRef`

Once you've established an association between two `ActorSystem` instances you can send messages back and forth between remote actors.

What makes all of this possible? The `RemoteActorRefProvider`!

The Role of `IActorRefProvider`s

All `ActorSystem` instances have exactly one `IActorRefProvider` internally - by default Akka.NET uses the `LocalActorRefProvider`. But in order to enable remoting, we had to change the `akka.actor.provider` property inside our HOCON configuration to use the `RemoteActorRefProvider` class. What does any of this mean?

An `IActorRefProvider` gives an `ActorSystem` the following capabilities:

1. Create new `IActorRef`s, which therefore means the ability to create actors;
2. Resolve `ActorSelection`s and figure out which actor should receive a message;
3. Deploy and configure `IActorRef`s based on anything specified inside the `akka.actor.deployment` configuration section; and
4. Start the `/system/` and `/user` guardian actors at the top of those actor hierarchies.

So in other words, `IActorRefProvider`s are *really important*.

What the `RemoteActorRefProvider` does is extend the capabilities of the `LocalActorRefProvider` to introduce the ability to:

1. Start all and configure all network transports;
2. Resolve `ActorSelection`s that are going out to or coming in from the network;
3. Create `RemoteActorRef`s for actors that exist elsewhere on the network;
4. Enable `DeathWatch` to work over the network and even detect network failures, which will be extremely useful to us later on; and
5. Support new deployment options that allow actors, routers, and routees of routers to

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

be deployed remotely onto other machines on the network.

So this is the central piece of plumbing that allows Akka.Remote to work. Now let's talk about how to use it!

Initiating Remote Communication via `ActorSelection`

There are only two ways to initiate a remote connection between two `ActorSystem`s:

1. Send a message to an `ActorSelection` on a remote system or
2. Deploy an actor remotely onto a remote system.

Let's talk about `ActorSelection` first.

As you learned in Akka.NET Bootcamp, `ActorSelection`s are a way to look up an actor by its `ActorPath` and deliver it a message.

Given that, we can use an `ActorSelection` to look up a *remote actor running on another machine*.

Here's an example from `AkkaTime`, the in-depth code sample in *Akka.NET Remoting* which we'll explain in the next major section, of where we use an `ActorSelection` to instantiate a remote connection between an `AkkaTime.Client` and `AkkaTime.Server`.

```
/* Runs inside AkkaTime.Client Program.cs */

// serverAddress is loaded as a string via configuration
var serverAddress = ActorSystemRefs.ServerAddress = Address.Parse(actorSystem.Settings.Confi

var handshakeActor =
    actorSystem.ActorOf(SessionHandshakeActor.OutboundProps(sessionManagerWrapper, session,
        (actorSystem.Settings.Config.GetTimeSpan("akkatime.sessions.handshake-timeout",
            TimeSpan.FromSeconds(4.0)))));

//subscribe the connectivity actor to all lifecycle events, including debug events
sessionManager.Tell(new SubscribeToSessionLifecycleEvents(serverStatus, true));

var remoteEndpoint =
    actorSystem.ActorSelection(new RootActorPath(serverAddress)/"user"/ActorPaths.ServerEndp
var connectTo = new ConnectTo(remoteEndpoint, BeginHandshake.Instance);
```

The key piece of code is this:

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
var remoteEndpoint =
    actorSystem.ActorSelection(new RootActorPath(serverAddress)/"user"/ActorPaths.ServerEndp
```

This is where the `ActorSelection` gets constructed from an `Address` we parsed from a HOCON `Config` object.

We could just as easily pass in a full string, however - something like this:

```
var remoteEndpoint =
    actorSystem.ActorSelection("akka.tcp://akkatime@127.0.0.1:14445/user/listen");
```

Both of these pieces of code are equivalent and will produce the same result in this case.

Replies and `RemoteActorRef`s

The `SessionHandshakeActor` running inside `AkkaTime.Client` will execute the following piece of code immediately following the example shown above.

```
/* Inside AkkaTime.Common.SecureActors.SessionHandshakeActor.cs */

/// <summary>
/// Initial state for outbound session actors
/// </summary>
private void Closed()
{
    _currentState = HandshakeState.Closed;
    Receive<ConnectTo>(to => to.Payload is BeginHandshake,to =>
    {
        to.RemoteDestination.Tell(new HandshakeInfo(_selfSession, _sessionManager.Manager, S
        Become(WaitForHandshake);
        Stash.UnstashAll();
    });

    ReceiveAny(o => Stash.Stash());
}
```

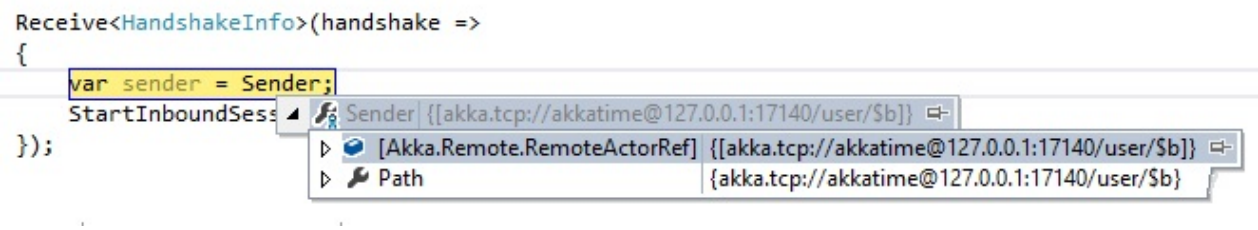
The `HandshakeInfo` message will be delivered over the network to a `SessionListener` actor,

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

who's running inside an `AkkaTime.Server` process. This is the code the `SessionListener` will run upon receiving this message.

```
public SessionListener(byte[] cryptKey, ServerSession activeSession, IActorRef authenticator
    IActorRef friendsActor, IActorRef presenceActor, IActorRef presenceSubscriber)
{
    _cryptKey = cryptKey;
    _activeSession = activeSession;
    _authenticator = authenticator;
    _registration = registration;
    _friendsActor = friendsActor;
    _presenceActor = presenceActor;
    _presenceSubscriber = presenceSubscriber;
    _autoAuthenticator = Context.ActorOf(Props.Create(() => new AutoAuthenticationApprover()

    _verifyTimeout = Context.System.Settings.Config.GetTimeSpan("akkatime.sessions.verify-ti
        TimeSpan.FromSeconds(2.0));
    _handshakeTimeout = Context.System.Settings.Config.GetTimeSpan("akkatime.sessions.handsh
        TimeSpan.FromSeconds(3.5));

    Receive<HandshakeInfo>(handshake =>
    {
        //who's the sender of this remote message?
        var sender = Sender;
        StartInboundSession(handshake);
    });
}
```

Something interesting happens when we receive this message. Check out the value of the `Sender` property!



Well, well, well! There's an `IActorRef` value for the `Sender` property alright - but it's a `RemoteActorRef` belonging to an actor that's not even running in the same process! Amazing!

So what happens if we `Tell` a message to this `RemoteActorRef`?

```
/* Inside AkkaTime.Common.SecureActors.SessionHandshakeActor.cs */
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
Receive<InboundSecureSessionInformation>(secure =>
{
    _remoteSessionHandshakeActor.Tell(new HandshakeAccepted(secure.SharedSecret, secure.Loca
        _sessionManager.Manager, _selfSession));
    _localSecureActor = secure.LocalSecureActor;
    BecomeWaitForSecureChannel();
});
```

The server also runs its own `SessionHandshakeActor` for each client who's connecting to it, and in this case it caches an `IActorRef` belonging to the `SessionHandshakeActor` on the `AkkaTime.Client` inside a field called `_remoteSessionHandshakeActor`.

After a bunch of authentication / identity verification stuff happens the server will hit the code block shown above and send the `SessionHandshakeActor` on the client the next message it expects as part of its message flow. Let's see what that looks like!

```
/* Inside AkkaTime.Common.SecureActors.SessionHandshakeActor.cs */

// OUTBOUND BEHAVIORS
Receive<HandshakeAccepted>(accepted =>
{
    //wonder who the Sender is?
    var sender = Sender;
    _remoteSecureActor = accepted.SecureActor;
    _remoteSession = accepted.Session;
    _remoteSessionHandshakeActor = Sender;
    _currentState = HandshakeState.Accepted;
   _sessionManager.StartSecureSessionOutbound(new OutboundCreateSecureSession(
            accepted.SharedSecret, accepted.SecureActor, accepted.SessionActor, accepted.Ses
});
```

So who is the `Sender` for this message, back on the `AkkaTime.Client`?

```
// OUTBOUND BEHAVIORS
Receive<HandshakeAccepted>(accepted =>
{
    var sender = Sender;
    _remoteSecureActo   Sender {[akka.tcp://akkatime@127.0.0.1:14445/user/listen/$c]}
    _remoteSession = ac   [Akka.Remote.RemoteActorRef] {[akka.tcp://akkatime@127.0.0.1:14445/user/listen/$c]}
    _remoteSessionHands   Path {akka.tcp://akkatime@127.0.0.1:14445/user/listen/$c}
    _currentState = HandshakeState.Accepted;
   _sessionManager.StartSecureSessionOutbound(new OutboundCreateSecureSession(
            accepted.SharedSecret, accepted.SecureActor, accepted.SessionActor, accepted.Session)).PipeTo(Self);
});
```

NO WAY! It's a `RemoteActorRef`! And it belongs to the `SessionHandshakeActor` running the

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

server.

`RemoteActorRef` and Location Transparency

What `RemoteActorRef` gives us is a magical property called location transparency

> What location transparency means is that whenever you send a message to an actor, you don't need to know where they are within an actor system, which might span hundreds of computers. You just have to know that actors' address.

It's the job of the `RemoteActorRef` to make a remote actor running on in a different process look and feel exactly the same as an `IActorRef` running locally inside the same process as your code. This is what it means to "have a transparent location."

Regardless of where the actor actually resides, it doesn't affect your code one way or another.

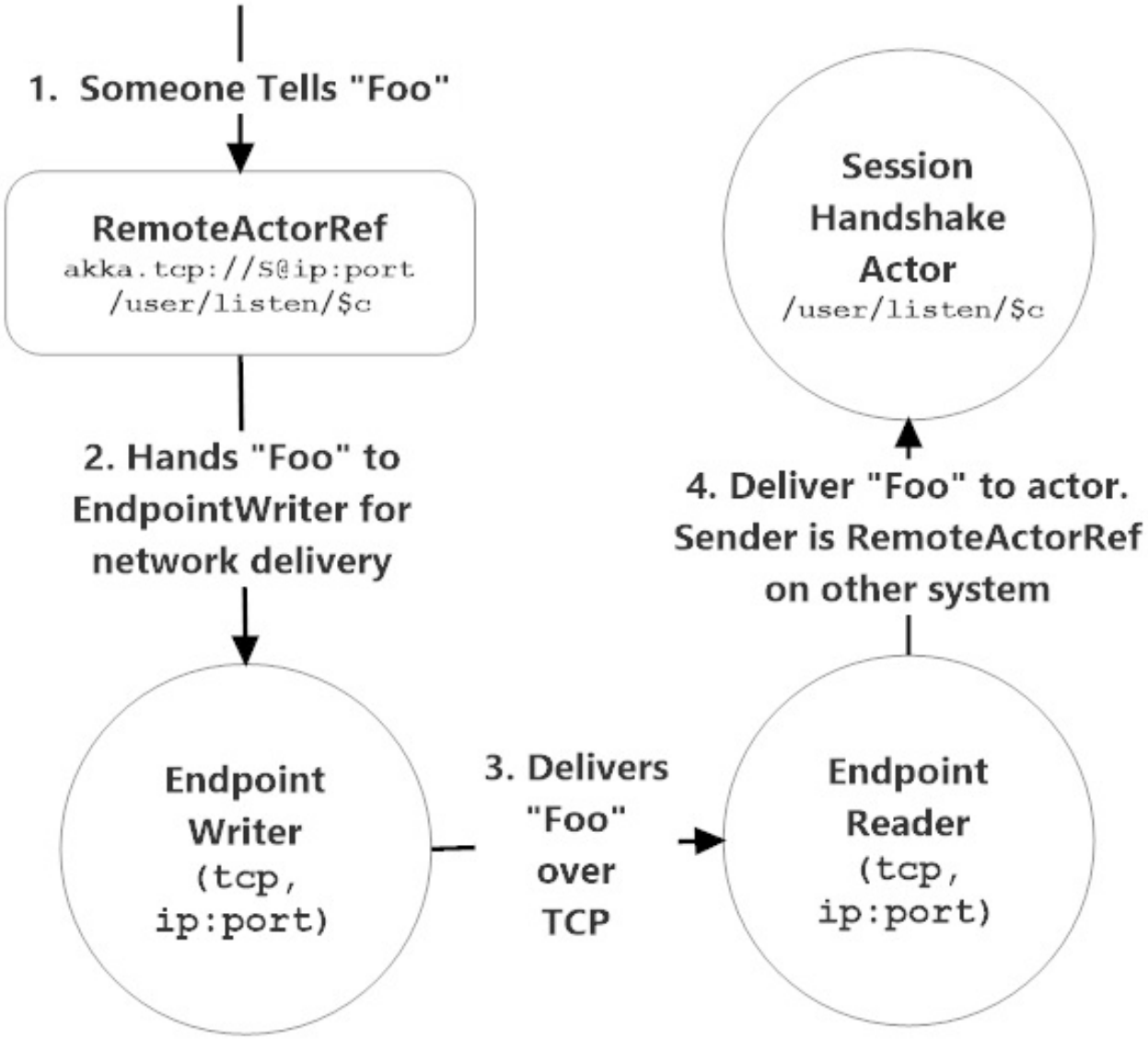
So this has one profound implicat on your Akka.NET applications - all of your actor code *is already able to run on the network by default*.

Therefore, many of the code samples in *Akka.NET Remoting* won't look very "networky." That's on purpose. That's Akka.NET taking care of the heavy lifting for us!

How `RemoteActorRef` Actually Works

So how does the `RemoteActorRef` class provide location transparency for us?

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



A `RemoteActorRef` is actually a paper-thin `IActorRef` instance that maps back to an actor that actually exists somewhere else on the network.

Every time you `Tell` a message to a `RemoteActorRef` you're actually writing a message to your local `EndpointWriter` for that transport, who will in turn deliver the message over the network to an `EndpointReader` on the other side of the association.

The `EndpointReader` is able to tell, with some help from the `RemoteActorRefProvider`, which local actor this network message was intended for and will deliver that message to the correct local actor.

And, as you might have guessed, it will also create a `RemoteActorRef` that the receiver of the network message can use the reply to the original sender!

So that is the essence of how `RemoteActorRef`s work - they're a tool to provide location

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

transparency and make it so all of your actor code that works for local actors can automatically work with remote actors without any code modifications.

Let's take a look at another way we can create `RemoteActorRef` instances - through remote deployment!

Remote Actor Deployment

One of the concepts that is unique to Akka.NET and the actor model is the notion of "actor deployment."

We explain how actors can be deployed in Unit 2.1 of Akka.NET bootcamp, so you should review that if you need a refresher on the fundamentals.

Deploying an actor means two things simultaneously:

1. Creating an actor instance with specific, explicitly configured properties and
2. Getting an `IActorRef` to that actor.

With Akka.Remote we get a new exciting detail: the **network location to which an actor is deployed becomes a configuration detail**.

That's right - we can *deploy code over the network* with Akka.Remote.

Here's what that concept looks like expressed as Akka.NET code:

Shared Actor / Message Code

```
/*
 * Create an actor and a message type that gets shared between Deployer and DeployTarget
 * in a common DLL
 */
/// <summary>
/// Actor that just replies the message that it received earlier
/// </summary>
public class EchoActor : ReceiveActor
{
    public EchoActor()
    {
        Receive<Hello>(hello =>
        {
            Console.WriteLine("[{0}]: {1}", Sender, hello.Message);
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            Sender.Tell(hello);
        });
    }
}

public class Hello
{
    public Hello(string message)
    {
        Message = message;
    }

    public string Message { get; private set; }
}
```

DeployTarget (process that gets deployed onto)

```
class Program
{
    static void Main(string[] args)
    {
        using (var system = ActorSystem.Create("DeployTarget", ConfigurationFactory.ParseStr
            akka {
                actor.provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
                remote {
                    helios.tcp {
                        port = 8090
                        hostname = localhost
                    }
                }
            }")))
        {
            Console.ReadKey();
        }
    }
}
```

Deployer (process that does deploying)

```
class Program
{
    class SayHello { }

    class HelloActor : ReceiveActor
    {
        private IActorRef _remoteActor;
        private int _helloCounter;
        private ICancelable _helloTask;
```

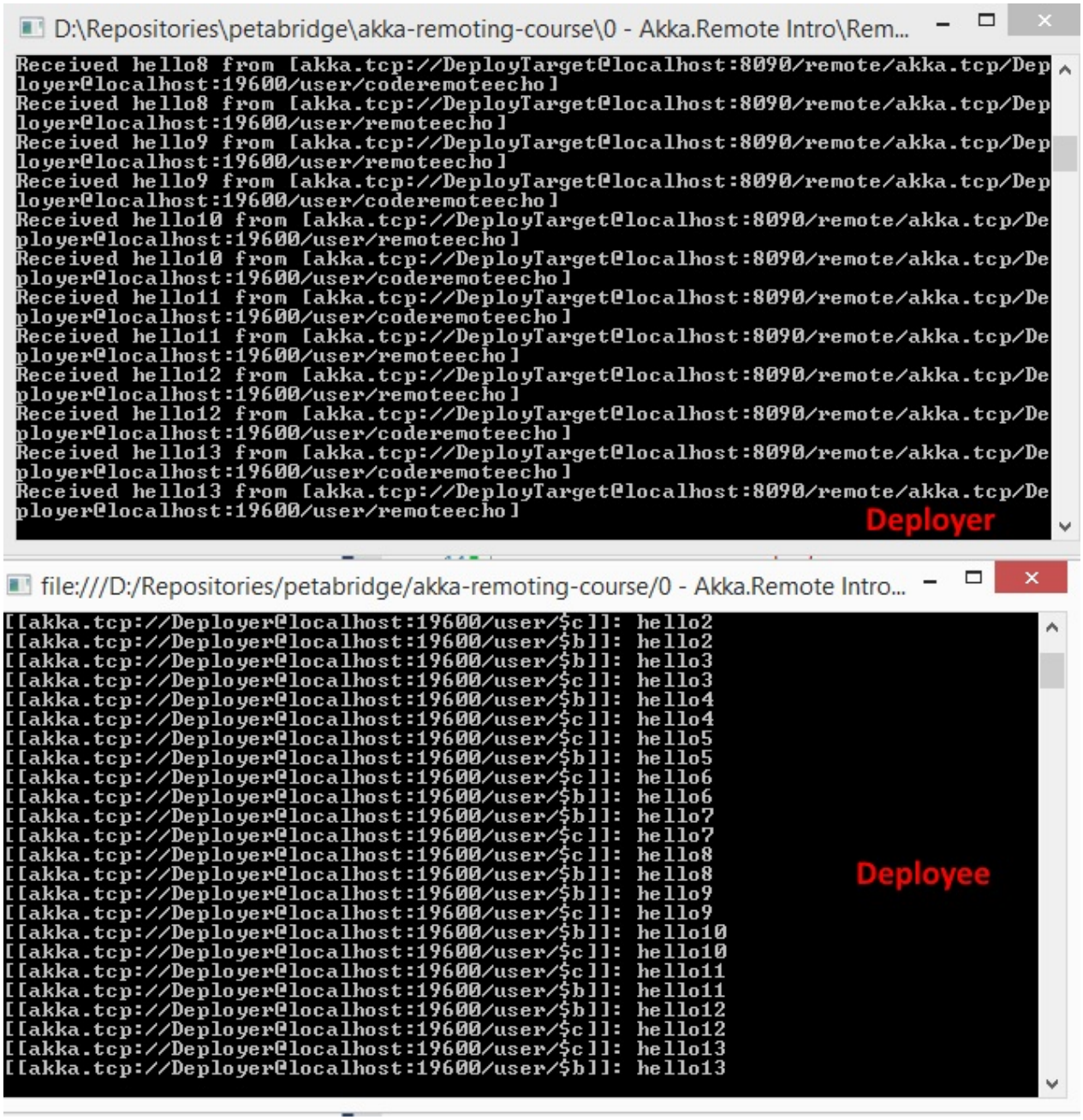

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        public HelloActor(IActorRef remoteActor)
        {
            _remoteActor = remoteActor;
            Receive<Hello>(hello =>
            {
                Console.WriteLine("Received {1} from {0}", Sender, hello.Message);
            });

            Receive<SayHello>(sayHello =>
            {
                _remoteActor.Tell(new Hello("hello"+_helloCounter++));
            });
        }

        protected override void PreStart()
        {
            _helloTask = Context.System.Scheduler.ScheduleTellRepeatedlyCancelable(TimeSpan.
                TimeSpan.FromSeconds(1), Context.Self, new SayHello(), ActorRefs.NoSender);
        }

        protected override void PostStop()
        {
            _helloTask.Cancel();
        }
    }

    static void Main(string[] args)
    {
        using (var system = ActorSystem.Create("Deployer", ConfigurationFactory.ParseString(
            akka {
                actor{
                    provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
                    deployment {
                        /remoteecho {
                            remote = ""akka.tcp://DeployTarget@localhost:8090""
                        }
                    }
                }
                remote {
                    helios.tcp {
                        port = 0
                        hostname = localhost
                    }
                }
            }")))
        {
            var remoteAddress = Address.Parse("akka.tcp://DeployTarget@localhost:8090");
            //deploy remotely via config
            var remoteEcho1 = system.ActorOf(Props.Create(() => new EchoActor()), "remoteech

            //deploy remotely via code
            var remoteEcho2 =
                system.ActorOf(
                    Props.Create(() => new EchoActor())
                        .WithDeploy(Deploy.None.WithScope(new RemoteScope(remoteAddress))),
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            system.ActorOf(Props.Create(() => new HelloActor(remoteEcho1)));
            system.ActorOf(Props.Create(() => new HelloActor(remoteEcho2)));

            Console.ReadKey();
        }
    }
}
```

So what happens in this sample? Well first, let's take a look at the output if we run this with just one deployer.



Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

As far as the **DeployTarget** is concerned, it's receiving messages from a `RemoteActorRef` belonging to the **Deployer**, hence why we can see `akka.tcp://Deployer@localhost:19600/user/$c` as the `Sender` address.

But wait! Wait a minute! What the hell?! We didn't actually create any actors in the DeployTarget process!!!!

That's because the Deployer created the actors... but it created them INSIDE DeployTarget's process. Over the network.

That's what this piece of HOCON configuration:

```
deployment {
    /remoteecho {
        remote = ""akka.tcp://DeployTarget@localhost:8090""
    }
}
```

And this piece of C# code:

```
Props.Create(() => new EchoActor()).WithDeploy(Deploy.None.WithScope(new RemoteScope(remoteA
```

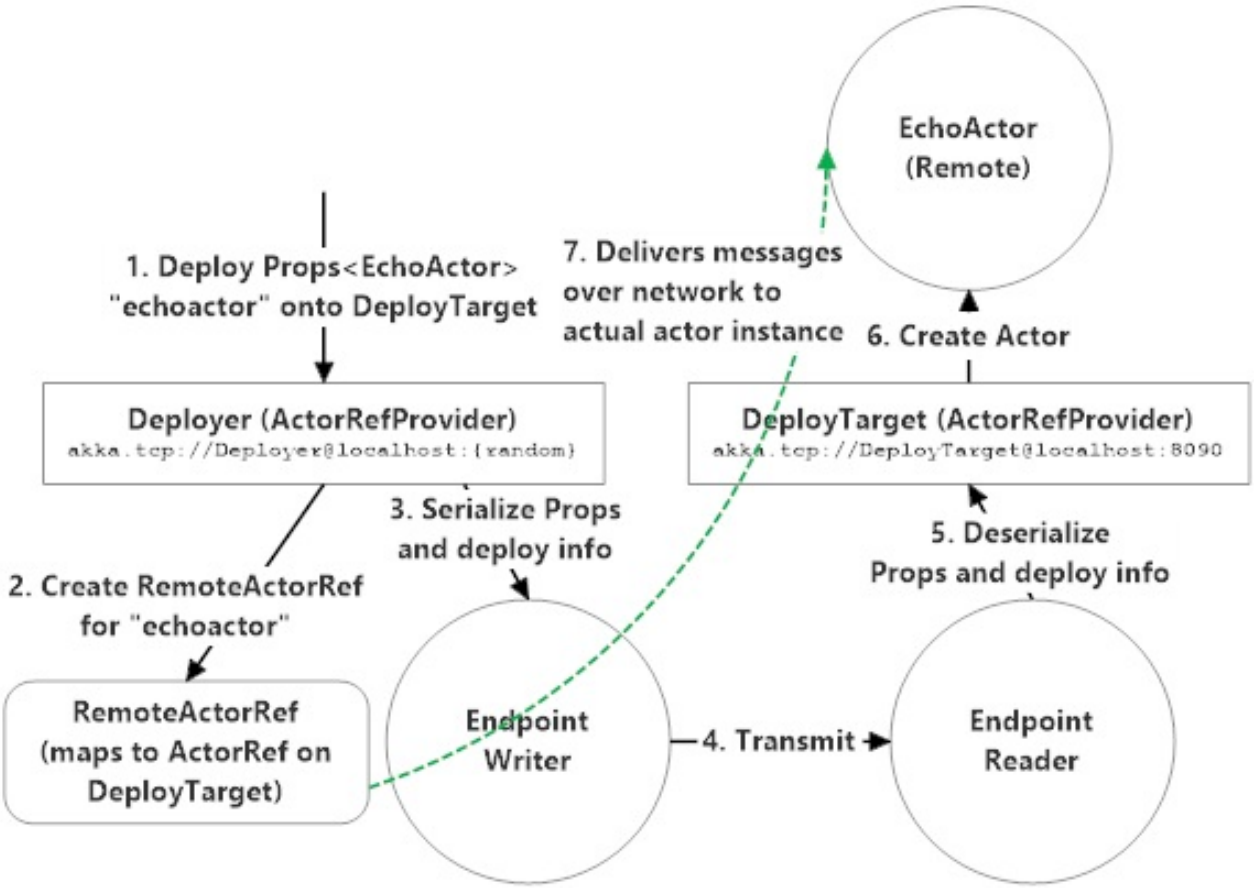
Both do respsectively. They deploy the configured actor onto the specified remote `Address`, which belongs to the DeployTarget in this case.

What is this new devilry?

How Remote Deployment Actually Works

The process of remote deployment feels magical, but it's actually pretty simple.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Rather than wait for a `RemoteActorRef` to send us a message, we create one in advance - knowing full well that we're going to be able to deploy one onto the remote actor system *if we're able to form an association with the remote system*. If we can't form a remote association with the DeployTarget system, then the `RemoteActorRef` we've just created will map to `Deadletters` and nothing will happen.

However, in the happy event that we can form a remote association with DeployTarget the following things will happen:

1. A local name on for the actor we're remote deploying will be reserved on Deployer, since all actors must have a unique name;
2. The `Props` for the `EchoActor` will be serialized, including all of the constructor arguments for the `EchoActor` class, along with anything else in the actor's deployment such as router details, dispatcher configuration, and so forth;
3. The serialized `Props` and all of the relevant actor path and name information is transmitted over the network by Deployer's `EndpointWriter` and received by DeployTarget's `EndpointWriter`;
4. The `EndpointWriter` determines that this is a special "deploy remote actor" message, and tells a special system actor (the `RemoteDaemon`) to create a new `EchoActor` instance; and

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

5. Going forward, all messages sent to the `RemoteActorRef` are automatically sent to the `EchoActor`.

And once all of that is done, we've successfully deployed an `EchoActor` over the network from Deployer to DeployTarget.

Important Things to Know About Remote Actor Deployments

We're not going to get into all of the details, because it involves some gory Akka.NET internals, but here are some important things to remember about remote actor deployments:

1. All names for remote actors are determined *by the deploying* `ActorSystem`. 1000 Deployer instances could all deploy an actor named "echoactor" onto the same DeployTarget instance and all 1000 of those operations would be successful. That's because the local actor created on DeployTarget has an `ActorPath` that looks like `akka.tcp://DeployTarget@localhost:8090/remote/akka.tcp/Deploye@localhost:19600/user/echoactor/` - the full `Address` of each Deployer `ActorSystem` is appended to the front of the `ActorPath`, thereby guaranteeing that each remote deployed actor name is unique to the Deployer.
2. The C# code that defines the `EchoActor` type and the message types it expects must be present on **both the Deployer and the DeployTarget**, or the deployment fails because the `Props` can't be deserialized.
3. All of the constructor arguments for `EchoActor` and any other remote-deployed actor must be serializable, again, because otherwise it can't be deployed.

And with that knowledge in-hand, we have all of the Akka.Remote basics covered!

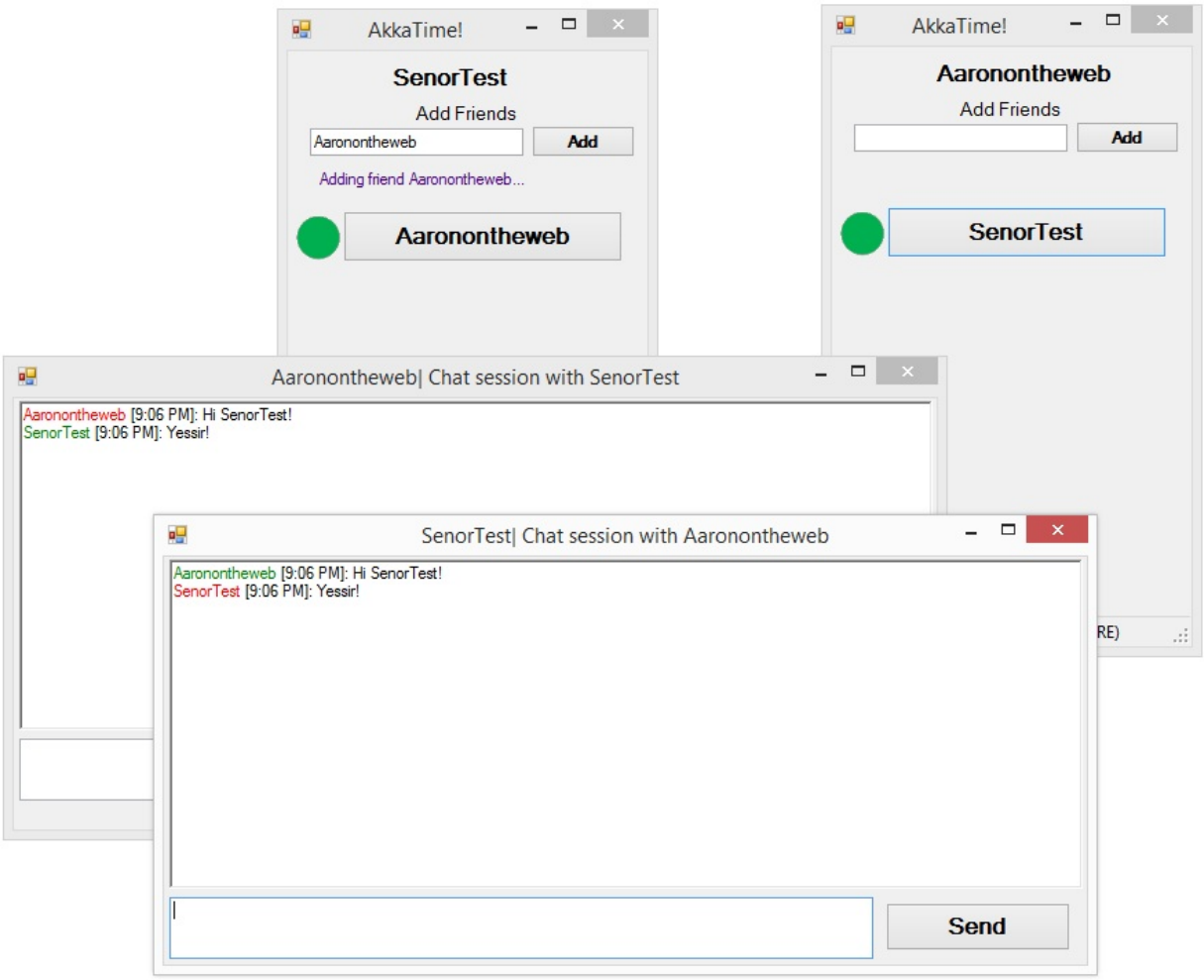
Next Step

Now that we've taken on the fundamental Akka.Remote concepts, let's explore our detailed **AkkaTime Akka.Remote Chat example** and use that as the basis for the rest of *Akka.NET Remoting*.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

AkkaTime - *AOL Instant Messenger* with Akka.NET

In *Akka.NET Remoting* we're going to look at the most ambitious code sample ever developed by the Petabridge team, AkkaTime!



AkkaTime Features

Before we dive into the architecture, consistency model, and code choices made in the design of AkkaTime, let's take a quick run through the features of AkkaTime.

- **Authenticated users and central account management** - there's a central server responsible for verifying and establishing the identity of each user, and only authorized users are allowed to chat with their friends;

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

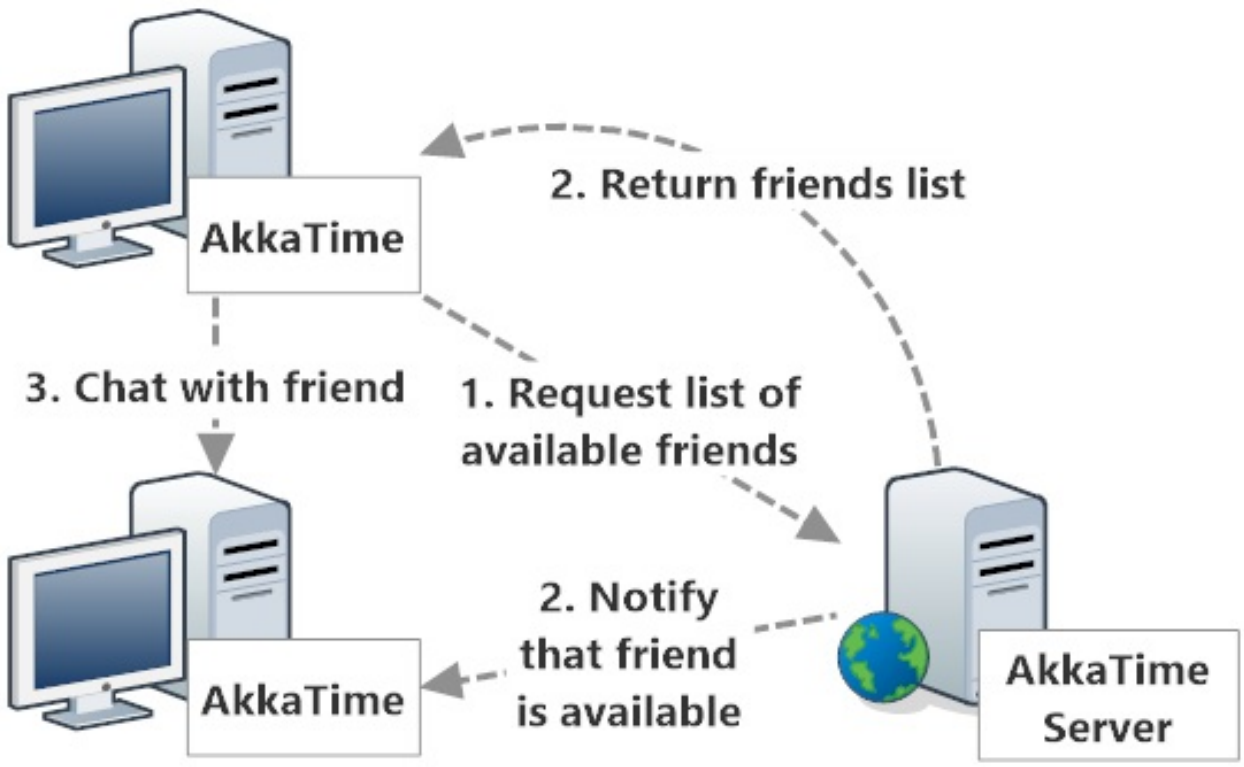
- **Point-to-point symmetric encrypted communication** - each connection, peer-to-peer or client-server, is secured using two way encryption with AES and HMAC;
- **Friends list and user presence** - each user has the ability to add friends and be notified when they're active;
- **Peer to peer communication** - each chat session is done via peer-to-peer connections, so no chat conversation ever goes through the central authentication server.

It's a pretty basic chat application, but it uses some excellent Akka.Remote design patterns to accomplish its goals and provide a strong consistency model across servers, clients, and peers.

AkkaTime Architecture

The AkkaTime architecture is a blend of both peer-to-peer and client-server architectures.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Conceptually, you can think of AkkaTime as two distinct applications that satisfy each other's message contracts over the network using Akka.Remote:

- `AkkaTime.Server` - a central server responsible for managing the identities, authentication details, presence information, and connectivity information for every registered AkkaTime user.
- `AkkaTime.Client` - a cross-platform Windows Forms application that authenticates with `AkkaTime.Server` in order to add friends and get access to presence and connectivity information about those friends. If an authenticated AkkaTime user has friends who are available, the AkkaTime client can establish a connection directly to those clients via peer-to-peer communication.

While most developers are familiar with client-sever architecture as a result of

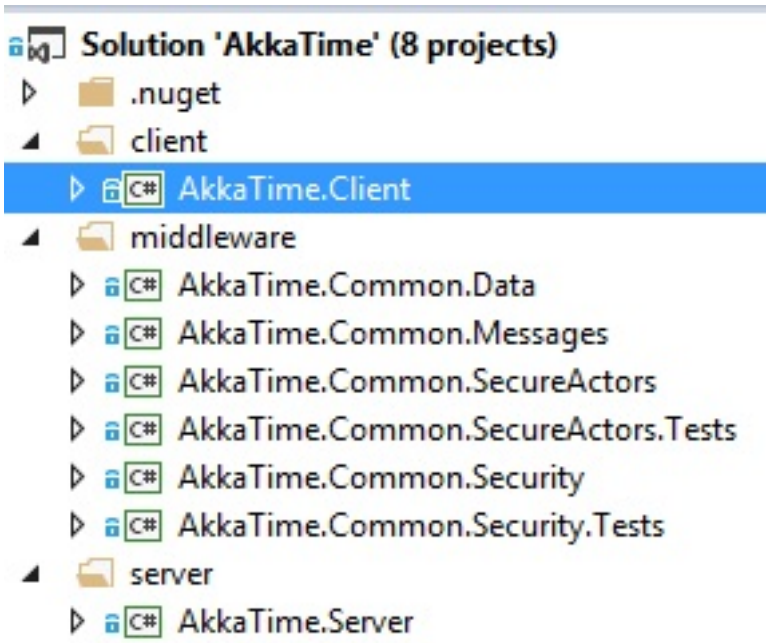
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

experience with web programming, many are not familiar with peer-to-peer networking. We made the decision intentionally to use peer-to-peer for the `AkkaTime.Client` interconnections because we thought it would serve as a good take-home example for how to use this yourself.

> NOTE: *Akka.NET Remoting* will cover network architecture patterns a little later on in this workbook.

Organization of the AkkaTime Solution

The AkkaTime solution is organized into three major areas:



- **Middleware** - actors, message types, and services that are shared across the AkkaTime project;
- **Client** - the code for the `AkkaTime.Client` application; and
- **Server** - the code for the `AkkaTime.Server` application.

Reusing Message, Actor Types

One of the core features of AkkaTime is secure, two-way encryption between client-server and client-client connections. Therefore most of the shared middleware in AkkaTime focuses on security.

The `AkkaTime.Common.Security` project exposes a number of service classes that expose some encryption and hashing primitives that run a bit beyond the scope of this course.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

The `AkkaTime.Common.SecureActors` project, however, is much more interesting. It contains some `ReceiveActor` implementations and associated messages that are used across both the `AkkaTime.Client` and `AkkaTime.Server` projects. Here are the important actor types to remember:

- **SessionHandshakeActor** - responsible for creating an encrypted connection between two endpoints on the network, after an association has formed.
- **TwoWayEncryptionActor** - uses the `AkkaTime.Common.Security.AESThenHMAC` class to encrypt `INeedEncryptionEnvelope` requests and decrypt `IEncryptedEnvelope` messages before forwarding the decrypted message onto its final destination.
- **SessionManagerBase** - an `abstract` actor that has implementations in both `AkkaTime.Client` and `AkkaTime.Server`; it's job is to act as the endpoint between any two AkkaTime processes, client-server, server-client, and client-client.

The other shared secure actor types are less important - these three are the ones that form the entire security apparatus and exposed endpoints for all AkkaTime applications over the network.

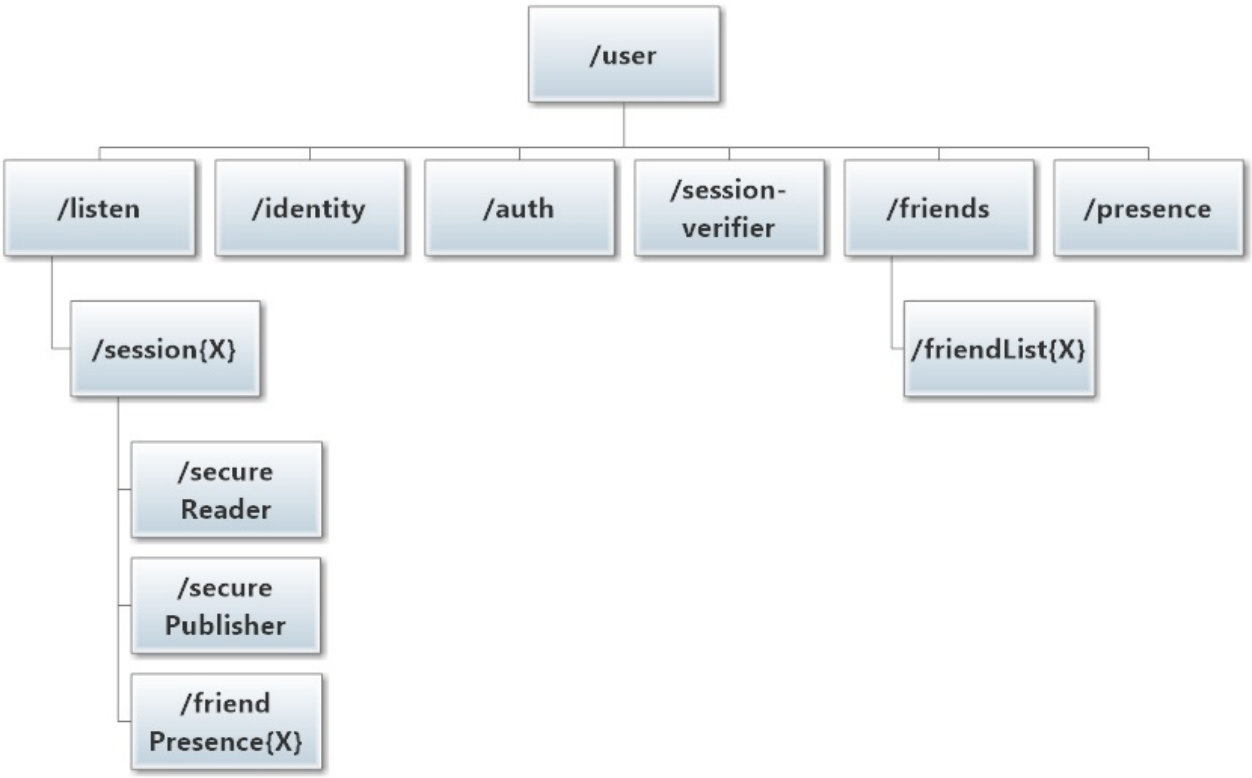
The message types that are shared vary in significance, but all of them are designed to be sent client-to-client or client-to-server.

> NOTE: It's possible to refactor some of these messages directly into the `AkkaTime.Client` project, but we left them in the shared library in case an attendee wants to rewrite the AkkaTime sample to be purely client-server or to support multiple types of AkkaTime clients, such as chat bots.

`AkkaTime.Server` Actor Hierarchy

The `AkkaTime.Server` hierarchy is relatively flat.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Most of the actors involved in running the `AkkaTime` server are a bunch of stateless routers and workers, such as the `IdentityStoreActor` who reads and writes `User` records to SQL Server and the `AuthChallengeActor` who processes password challenge requests from clients.

The important actors are as follows:

1. **SessionListener** (`/listen`) - this is the only actor who's `ActorPath` is known to all `AkkaTime.Client` instances, so it serves as the inbound point of contact for all `AkkaTime.Client` processes when they first make contact with the `AkkaTime.Server`.
2. **ServerSessionManager** (`/listen/session{X}`) - precisely one `ServerSessionManger` is created per client connection. It's job is to secure the connection with the client before password / authentication information is exchange. From that point onward, once the user is authenticated, it's the job of the `ServerSessionManager` to act as the go-between for all other requests the client can make of the server.
3. **PresenceActor** (`/presence`) - there's only one instance of the `PresenceActor`. It's job is to act as the single source of truth for determining the online / offline / idle status of all AkkaTime users. It never speaks directly to `ServerSessionManager`s or remote clients of any kind.
4. **FriendListActor** (`/friends/friendList{x}`) - this is a `PersistentActor` (Akka.Persistence) that uses an internal event source to manage the friend list for an individual user. The `FriendsMaster` is the parent of this actor, and it bootstraps

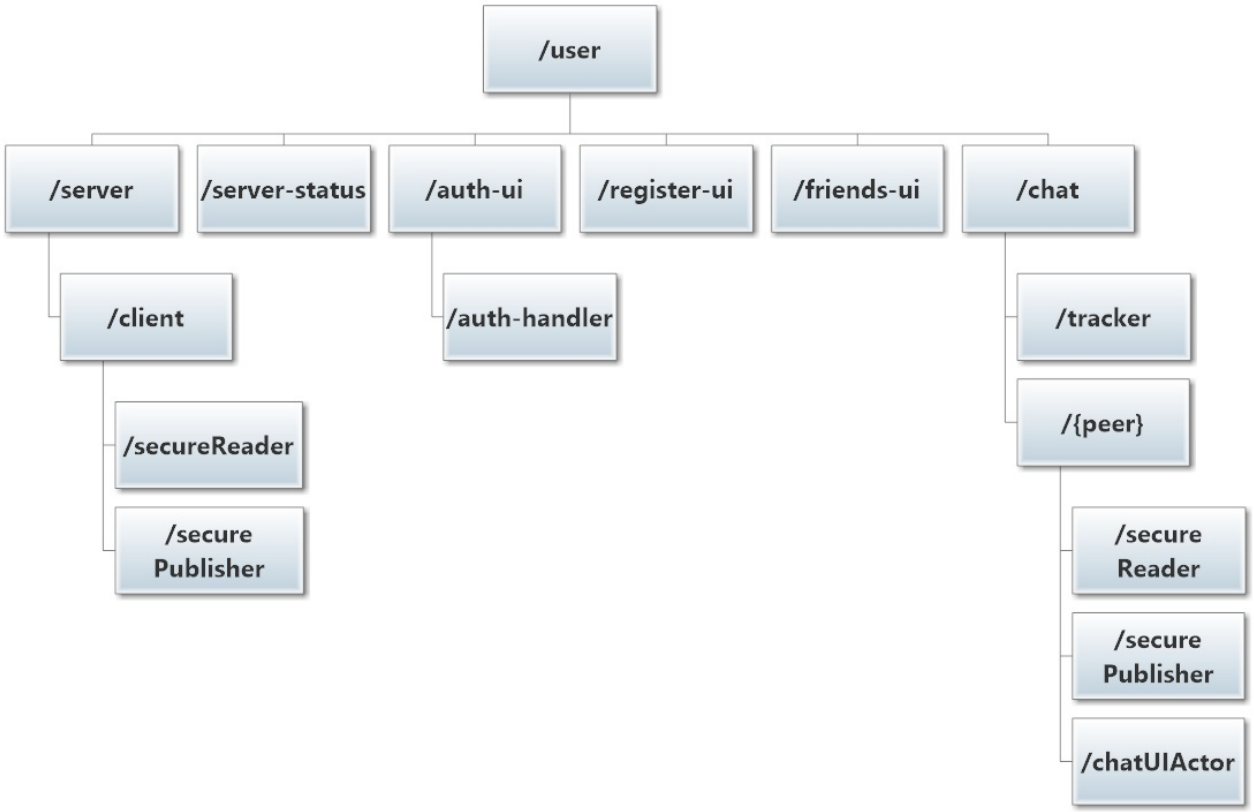
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

exactly one `FriendListActor` per user each time (Actor-per-Entity pattern, for those who have done the *Akka.NET Design Patterns* training) the server starts.

5. **FriendPresenceActor** (`/listen/session{X}/friendPresence{X}`) - this actor uses the `SecurePublisherActor` to communicate directly with a `FriendsUIActor` running on an `AkkaTime.Client` instance. It subscribes to updates from the `PresenceActor` and in turn forwards those updates to the `FriendsUIActor` on the client.

`AkkaTime.Client` Actor Hierarchy

The `AkkaTime.Client` actor hierarchy is a little more diverse, given that the client is responsible for UI in addition to both client-server and client-client state.



There are lots of actors who operate as just go-betweens for the Windows Form UI and the state of the client, but most of the actors in `AkkaTime.Client` act as subscribers for changes in session state between the server or another a client (peer.)

Here are the key actors that run the show inside `AkkaTime.Client` :

1. **ServerConnectionSupervisor** (`/server`) - this actor has one job: to start and restart the connection and session establishment process with the server until we are able to successfully reconnect. It even has the ability to silently re-authenticate with the

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

server in the event of an intermittent server failure.
2. **ClientToServerSessionManager** (`/server/client`) - this actor acts as the writable endpoint for sending messages to the server. There exists exactly one `ClientToServerSessionManager` for every `ServerSessionManager` on the server, as they form an exclusive, authenticated pair. It's also the job of this actor to work with the `AuthenticationUIActor` to help let it know when the session has been authenticated with the server and we can now show the "friends" UI.
3. **FriendsUIActor** (`/friends-ui`) - this actor subscribes to user presence updates from the `AkkaTime.Server.FriendPresenceActor` and allows new chat sessions to be attempted with users who are marked as "online" or "idle." This actor is also responsible for passing the `Address`es of remote peers (published by the `FriendPresenceActor`) with the `ChatLauncherActor` to attempt to launch new chat sessions.
4. **ChatHandlerActor** (`/chat`) - this actor is responsible for initiating connections with other `ChatHandlerActor`s running on other authenticated `AkkaTime.Client` instances. It works with the `ChatSessionsTracker` to make sure that there's a maximum of one current chat session per user/user pair. It also receives updates including new authentication credentials from the `ServerConnectionSupervisor` in the event of a server restart.
5. **ClientToClientSessionManager** (`/chat/{peer}`) - this actor is responsible for creating secure sessions between peers and creating all of the plumbing necessary to pass encrypted messages back and forth. It also (optionally, tunable via HOCON configuration) communicates with the `AkkaTime.Server.SessionVerificationActor` to verify that inbound chat requests are, in fact, from authenticated users.

Next Steps

We'll be referring to AkkaTime throughout the next sections of *Akka.NET Remoting*. AkkaTime is a large project, so don't worry if you get lost in some of the names and code right away. That's a good thing and it gives you something to work on later, piecing together how it works. But it's a well-designed, ambitious, application *that can still be greatly improved.*

In the next section we'll talk about **Remote Actor Communication and Deployment in Practice**

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Remote Communication and Deployment

Now that we've covered the fundamentals of Akka.Remote, and how AkkaTime works, it's time to move to a broader question: *how do you design an actor hierarchy for an Akka.NET application that works over the network?*

We'll address that in this section, and then close this section with a tour of some solid principles / best practices for writing your code now that the network is woven throughout the application.

Designing Across Network Boundaries

When designing your application to run via Akka.Remote, most of your code will be shaped by two questions:

1. How do you break up actor hierarchy across different apps, or cloned instances of the same app? (We'll refer to each of these as a "process" regardless of which you choose.)
2. Do you communicate between processes by looking up existing actors to talk to, or do you remotely deploy new actors and then communicate with those resources you just remotely deployed?

Let's explore these now.

Designing Your Actor Hierarchy

The Cardinal Rule of Akka remote

If there is one rule to remember from this section of the course, it's this:

> **Location transparency is everything.**

If you keep this in mind, you will usually make the right choices in structuring your code.

Why Location Transparency Matters So Much

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Location transparency is critical because it makes it simple to transition an actor system from local-only, to working remotely across the network. The idea that "location is a configuration detail" is essential to the ability to develop a multi-process `ActorSystem` locally, and have it work as intended once distributed across a network of machines.

Without location transparency, you would be constantly rewriting your code to reflect where the code was going to run.

With location transparency, you can write your code on one machine, and scale it out to many by changing configuration rather than rewriting your core actors.

So how do we model our hierarchies with this in mind, and actually bring location transparency to our `ActorSystem`?

How to Achieve Location Transparency

Limit Visibility into Hierarchies

Your actors in one hierarchy should know very little about any remote hierarchy that they collaborate with. Ideally, your actors should be developed so that it is not apparent that the actors they collaborate with are remote actors at all.

> **Rule of thumb**: If you can't tell whether an actor is local or remote by looking at your code—i.e. there are no hardcoded addresses, just lots of interactions between `IActorRef`s—your approach is probably right. If your actors are written to collaborate with *specific actors at specific addresses*, your approach is probably wrong.

If you write your actors to work with specific collaborators, you are breaking location transparency.

If you can't tell whether an actor is local or remote just by looking at the code, you're on the right track. This is a sign that your code is embracing location transparency.

Each application should have very limited knowledge about the other actor hierarchy, even if they're two copies of the same application. You don't want tight coupling between actor hierarchies in two or more applications.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

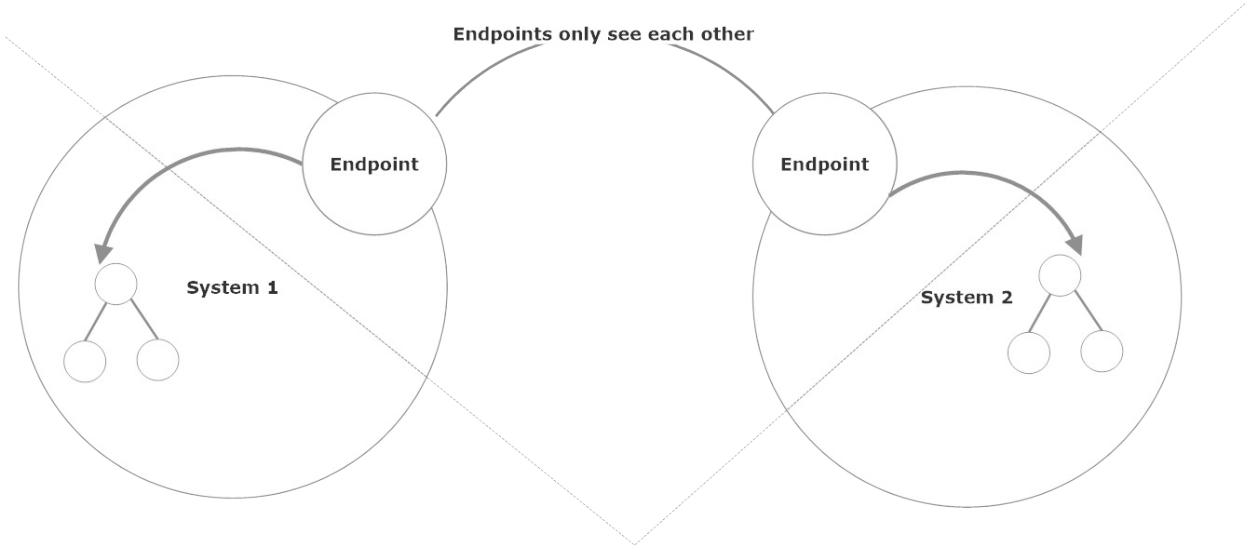
The exception to this rule occurs with what we call "contact points," or the endpoints that form an association between `ActorSystem`s.

Only the Endpoints Should Know About the Association

So if we need to limit what our actors in one hierarchy know about actors in another remote hierarchy, how do we connect them at all?

The answer is by isolating the knowledge of remote processes, and the link to those processes, to your Endpoints.

At a high level, this is what it looks like for endpoints to be the only actors that actually know about the remote process:



As shown in this diagram, you can imagine an invisible wedge between the local actor hierarchy in a given process and the remote actor hierarchy it collaborates with. If done right, it should not even be apparent from the actor code that the two hierarchies are separated by the network.

In practice, this means that contact point actors have an additional responsibility besides establishing the link between processes: Contact points, as the gateways between remote processes, are responsible for making sure all the other actors in the local hierarchy don't know they're working with `RemoteActorRef`s. The local actors should not know or care that they are working with remote actors.

> **Rule of thumb**: the contact points should make the network invisible to their local actor hierarchies.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

In order for actors to work with their remote counterparts transparently, `RemoteActorRef`s need to be spread throughout the hierarchies. This means that it is the responsibility of the dedicated "endpoint" actors to distribute the `RemoteActorRef`s around as appropriate. If done right, those local actors won't even know they're working with remote actors-the actors can just reply to their `Sender` and/or pub/sub across the network without even knowing it.

What this ends up looking like depends totally on the messaging model. But this approach is how you can make sure that the `RemoteActorRef`s cross network boundaries *and* preserve location transparency in your actors at the same time.

Make it so only the endpoints know about the remote process and make the network between actors invisible.

> Incidentally, this is another good reason not to use `ActorSelection`s unless you need to: you can't pass an `ActorSelection` around and get reliable results, since its path can be relative. And when working with remote Akka.NET processes, you *need* the ability to pass around `IActorRef`s without concern for where the actors actually live. Again: *Location transparency is everything.*

There is one more practice needed to make the network invisible to local actors: messaging contracts.

Contact Points Need Messaging Contracts

Rather than starting by thinking about the specific actors that are sending and receiving messages across the network, start modeling the link between remote processes by thinking in terms of the message flow between your endpoints. If you've taken our Design Patterns course, this will sound familiar: it's what we referred to in that course as "modeling concerns as conversations."

In practice, what this means is that the endpoint actors in your processes should work within a well-defined messaging contract. This message flow should clearly lay out what sequences and types of messages each endpoint expects to receive from its counterpart, wherever that counterpart may be on the network.

Code Sample

Here's an example from `AkkaTime` that demonstrates the additional responsibility contact

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

points have for making the network invisible to their local actors, and the messaging contract that achieves it.

Here are the actors involved, and their roles:

- `ClientToServerSessionManager`: this is the client-side endpoint / contact point actor
- `ServerSessionManager`: this is the server-side endpoint / contact point actor
- `FriendsUIActor`: this is the local, client-side actor that receives updates over the network from the server, without any knowledge that the messages come from across the network
- `PresenceActor`: this is the local, server-side actor that is the single source of truth for what users are/aren't online. This actor publishes out changes in user presence to all of its subscribed `IActorRef`s, without knowing or caring if those subscribers are in-process or remote.

And now, the code:

```
// in FriendsUIActor.cs
protected override void PreStart()
{
    // Kicks off a process that has the server start subscribing this actor
    // to changes in friend availability and presence
    _clientActorRef.Tell(new BootstrapFriendsPresence(Self));
}

// in ClientToServerSessionManager.cs
/// <summary>
/// Authenticated behaviors
/// </summary>
private void Authenticated()
{
    // This request can only come from a friends UI actor
    Receive<BootstrapFriendsPresence>(bootstrap =>
    {
        var sender = _friendUIActor = Sender;
        // this message is received by the the ClientToServerSessionManager
        // and sent to the server endpoint, without the FriendsUIActor knowing
        _securePublisher.Tell(new SecurePublish(RemoteSessionActor, sender, bootstrap));
    });
    // ...
}

// in ServerSessionManager.cs
private void AuthorizedReceives()
{
    Receive<BootstrapFriendsPresence>(friends =>
    {
        if (_userPresenceManager == null)
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        {
            var userName = GetRemoteSessionInformation().UserName;
            _userPresenceManager =
                Context.ActorOf(
                    Props.Create(
                        () =>
                            new FriendsPresenceActor(GetRemoteSessionInformation().UserName,
                                _presenceActor, _friendsActor)));
            // upon receiving BootstrapFriendsPresence message, the server endpoint
            // subscribes the client-side FriendsUIActor to the server-side PresenceActor
            // invisibly, without either knowing about the other directly, or that there
            // is a network connection between them!
            _userPresenceManager.Tell(new SubscribeToMyFriendsPresenceChanges(userName, frie
            _userPresenceManager.Tell(new FetchAllPresenceInformationForMyFriends(userName))
        }
    });
    // ...
}
```

In this sample, the `FriendsUIActor`-responsible for displaying the list of friends and their online/offline status-is *leveraging a remote actor without knowing it*. In `PreStart()`, the `FriendsUIActor` is sending a `BootstrapFriendsPresence` message, with itself as the argument, to the `PresenceActor` on the `AkkaTime.Server`, but it has no idea.

As far as the `FriendsUIActor` knows, its telling its local endpoint-the `ClientToServerSessionManager` that is stored in the `_clientActorRef` field-to bootstrap the info about friend presence and subscribe the `FriendsUIActor` to updates. But this subscription to presence updates is actually going across the network to the remote `PresenceActor` on the server, via the `ServerSessionManager`. Similarly, the `PresenceActor` on the servers doesn't know that the `FriendsUIActor` is a remote subscriber of what it publishes. The `PresenceActor` just sends messages to its `IActorRef` subscribers, without any concern for whether those subscribers are local or remote.

Message flows like this are important throughout the `ActorSystem`, but are absolutely critical between the dedicated endpoints make the network bridge between processes invisible. This is a great example of endpoints—`ClientToServerSessionManager` and `ServerSessionManager`—making the network link invisible to the local actors in each of their processes.

Choosing Between Hierarchy Designs Strategies

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

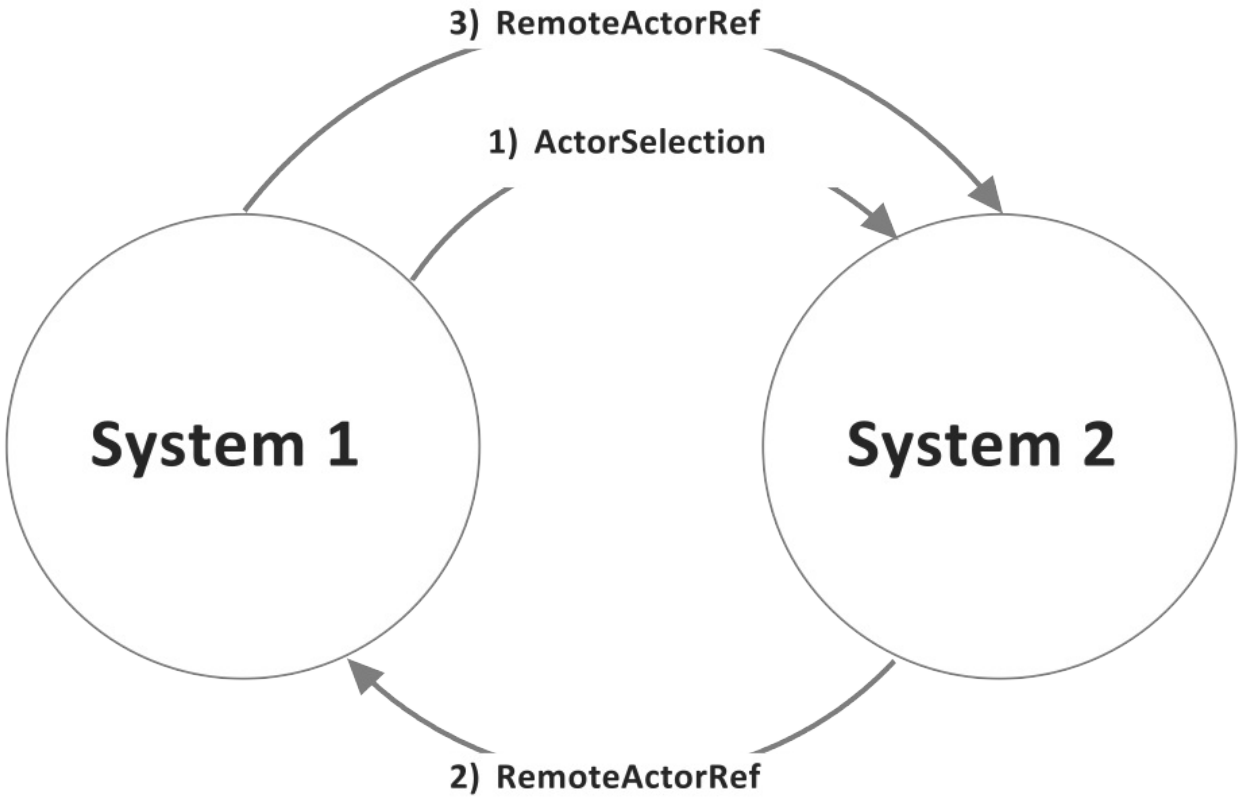
There are only two approaches to initialize a remote connection in Akka.NET: remote lookup and remote deployment

Let's cover what they are and when to use each.

Remote Lookup

Remote lookup is the process of using a remote `ActorSelection` to establish communications with a remote system. Essentially, you are "looking up" the remote actor you need to communicate with to open the link.

Here is a diagram of remote lookup. Note the conversion from `ActorSelection` to `IActorRef`:



In the above diagram, imagine processes `A` - `F` each "looking up" the endpoints on their remote counterparts, completing the messaging contract to connect the processes, and then collaborating over the socket.

The pattern you want to follow is to use remote lookup via an `ActorSelection` to establish the link between actor systems, and then transition to using `IActorRefs` exclusively after the endpoints have completed their messaging contract and bootstrapped the link between processes.

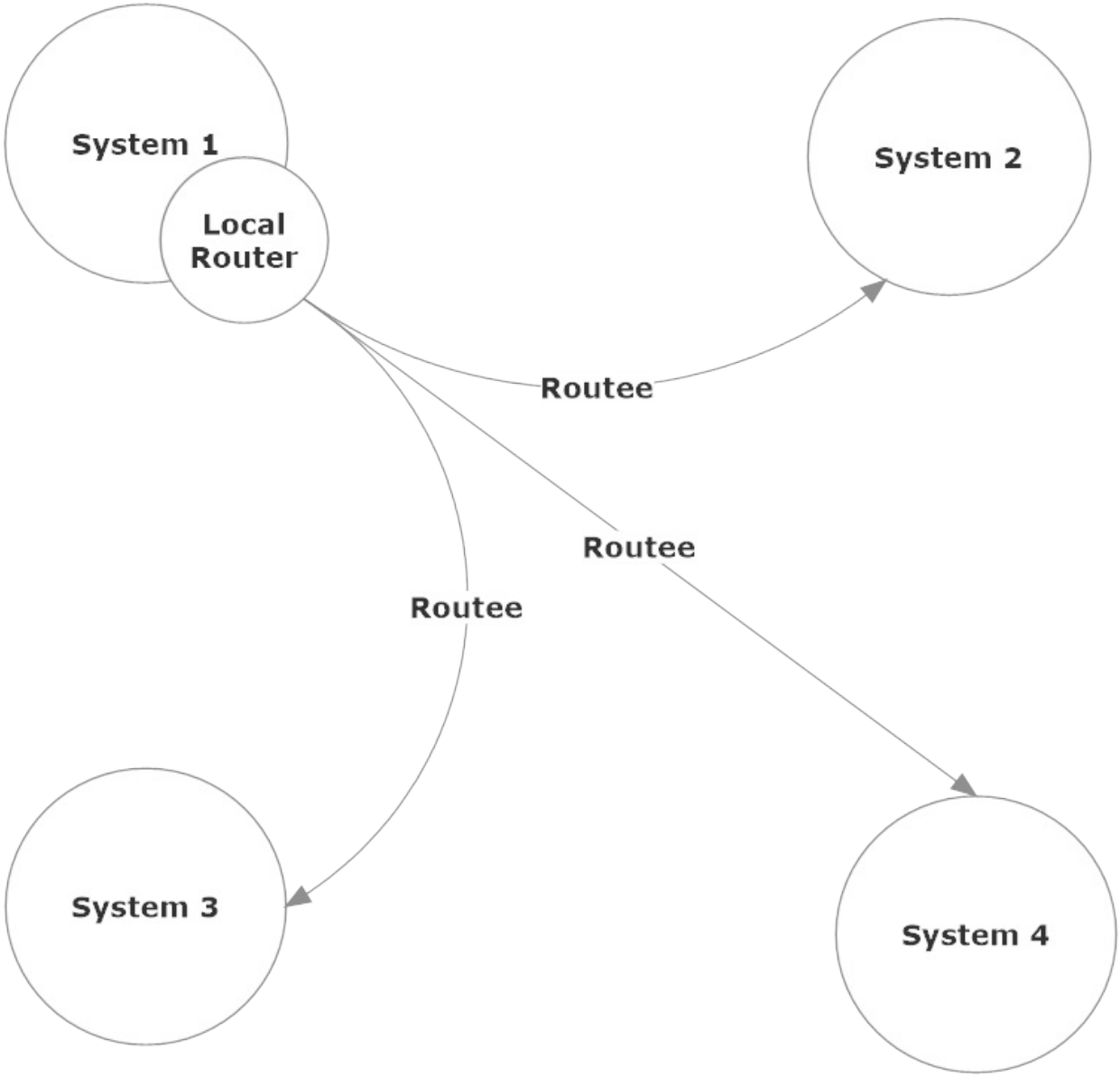
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Remote lookup is what `AkkaTime` uses, because it is a complex domain and actors on each process follow a messaging protocol to link up with their counterparts.

Remote Deployment

Remote deployment is a bit different: rather than trying to make initial contact with an existing remote actor, with remote deployment you are "pushing" the actors you need from your local process into a remote process.

Here's a visualization of remote deployment. In this diagram, a local `ResizablePoolRouter` is remotely deploying new routees to scale its worker pool across multiple remote machines:



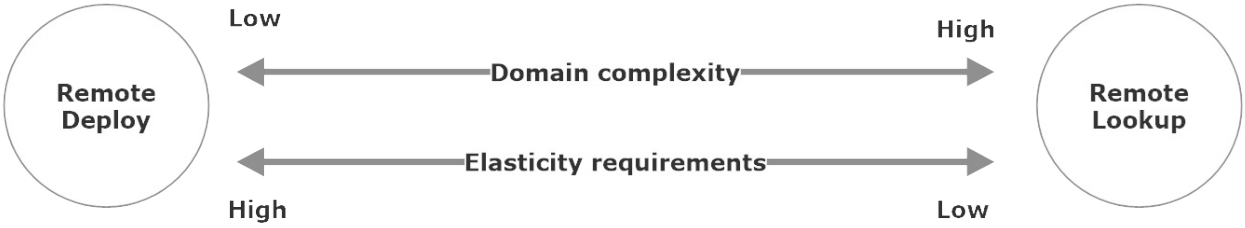
Generally, remote deployment is used as a work distribution strategy when you need to

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

farm out work across nodes. In the above diagram, you can see the common pattern of a local router "pushing" its routees (or even entire processing hierarchies) onto remote processes.

Choosing Between Lookup & Deployment

The choice between lookup and deployment comes down to the requirements of your system. It's not an either or choice (you can mix) but generally, the choice falls along a gradient between complexity of your domain, and its elasticity requirements:



Are you trying to link complex, remote actors? Use lookup. Trying to dynamically scale out the workload on your existing system across process and machine boundaries? Use deployment.

Remote lookup favors complex domains with low elasticity requirements. Most applications fall into this end of the spectrum-they have well-defined but complex components that need to interact, but don't necessarily have a high need for elasticity.

Remote deployment favors simple domains with high elasticity requirements. This strategy is essentially borrowing the CPU and memory resources of a remote machine to complete some task. A good rule of thumb is that if you're thinking of using PaaS autoscaling for a service, that service is probably more in the deployment end of the spectrum.

Remote deployment is much less common than lookup, although it is an amazing strategy to have in your toolkit when you need it. `AkkaTime` is not a good fit for remote deployment and so does not use it, but the scalable WebCrawler demo is a perfect fit.

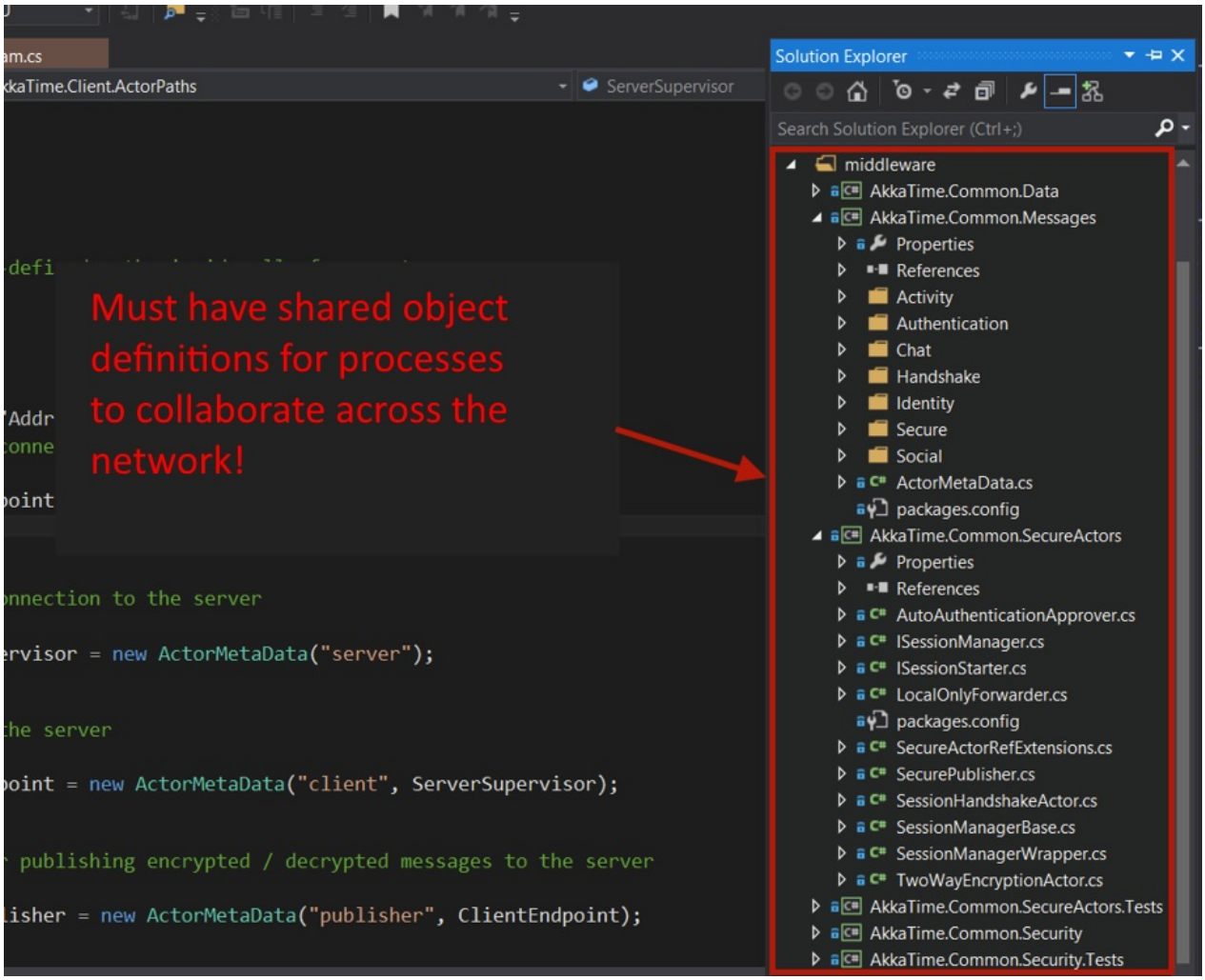
Best Practices & Things to Watch Out for

Shared Object Definitions Are Required

This is one point that is so obvious, it's easy to overlook.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Each side of network needs to have shared object definitions for your actor types, and any other objects you want to be shared across process boundaries. Otherwise, the correct actors can't be deployed remotely and everything breaks.



This is a good place to fold in the practice of centralizing `ActorPath`s and `ActorMetaData`, to 1) avoid errors fat-fingering in `ActorPath`s, and 2) make it easy to get commonly needed data about the actors from the objects you serialize and pass around. This is covered in depth in our Akka.NET Helpers repo, but you can see it being applied in the `AkkaTime` sample code like so:

```
using AkkaTime.Common.Messages;

namespace AkkaTime.Server
{
    /// <summary>
    /// Contains lookup paths and other relevant information for our server-side actors
    /// </summary>
    public static class ActorPaths
    {
        /// <summary>
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        /// Top-level actor for all things identity-related.
        /// </summary>
        public static readonly ActorMetaData Identity = new ActorMetaData("identity");

        /// <summary>
        /// Top-level actor for managing user-presence
        /// </summary>
        public static readonly ActorMetaData Presence = new ActorMetaData("presence");

        /// <summary>
        /// Actor responsible for managing session lifecycle subscriptions for the presence
        /// </summary>
        public static readonly ActorMetaData PresenceSubscriptions = new ActorMetaData("pres

        /// <summary>
        /// Authentication actor, responsible for handling auth challenges
        /// </summary>
        public static readonly ActorMetaData Auth = new ActorMetaData("auth");

        /// <summary>
        /// The primary endpoint for all clients to connect
        /// </summary>
        public static readonly ActorMetaData SessionListener = new ActorMetaData("listen");

        /// <summary>
        /// Used to verify the identities of clients during peer-to-peer connectivity
        /// </summary>
        public static readonly ActorMetaData SessionVerifier = new ActorMetaData("validator"

        /// <summary>
        /// Registration endpoint
        /// </summary>
        public static readonly ActorMetaData UserRegistration = new ActorMetaData("registrat

        /// <summary>
        /// Belongs to the friends master
        /// </summary>
        public static readonly ActorMetaData Friends =new ActorMetaData("friends");
    }
}
```

Serialization Gotchas

As with anything involving the network, serialization is at the heart of Akka.Remote. In Akka.NET, messages are byte arrays serialized to a string using JSON.NET, and are then packed into an envelope that uses the Google protobuf wire format. This envelope contains metadata about the message, such as the message size, sender, recipient, etc.

Fortunately, serialization (docs) is not something you have to manage as the developer.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

The framework handles it for you automatically, although you can implement a custom serializer if you choose.

So what do you need to think about with regard to serialization? The rule is simple: **Make your objects easily serializable**. This applies to your `Props` objects as well!

The most common sources of errors with serialization are when developers try to remote-deploy the following:

- resources that are specific to one machine (e.g. a performance counter, network connection, file handle, database connection)
- actors that depend on dependency injection for their functionality
- actors that takes an Action or Func as a constructor argument
- an actor type that takes an non-shared implementation of a shared interface, or a non-shared subclass of a shared class (Akka.NET's serialization is not tolerant of type ambiguity)

Always Reply-to- `Sender`

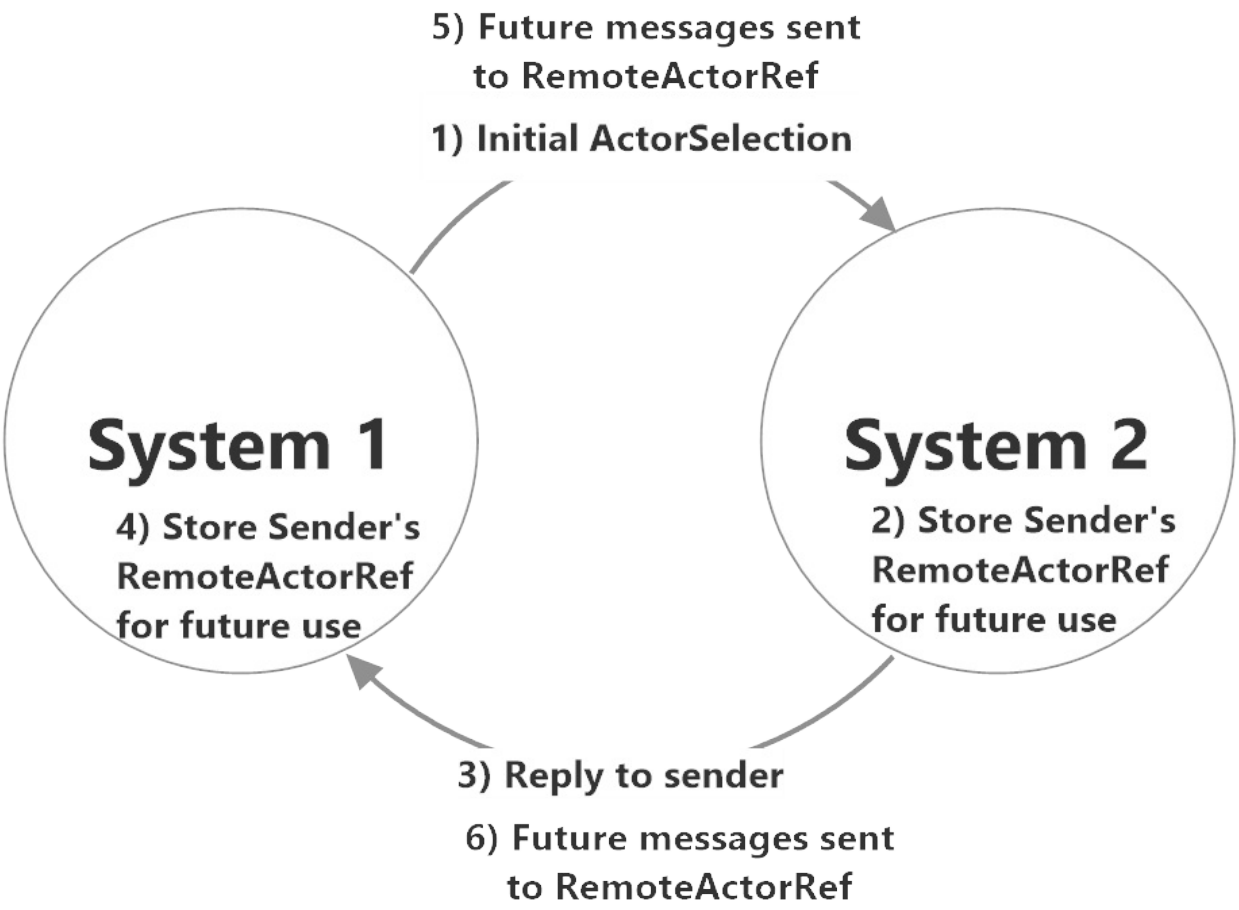
You want to be communicating with and passing around `IActorRef`s, not ActorSelections. In the context of a remote lookup, what this means in practice is that after establishing the first contact between systems, you want to convert from an `ActorSelection` to an `IActorRef` almost immediately, and then use the `IActorRef`s for all ensuing communications. **Convert from `ActorSelection` to `IActorRef`, ASAP.**

The right way to do this is very simple: on either side of the association, just reply to the `Sender` of the incoming messages and find a way to store that `IActorRef` so you can re-use it.

Rule of thumb: you should always reply-to-sender when you're expecting to receive messages from a remote system.

Here's what that looks like:

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



// session handshake actor => replying to handshake actor across the network

And here's an example from `AkkaTime`, where the `SessionHandshakeActor` replies across the network to a remote copy of itself in order to continue the handshake process that secures communications (more on security in the next section):

```
// SessionHandshakeActor.cs
// OUTBOUND BEHAVIORS
Receive<HandshakeAccepted>(accepted =>
{
    _remoteSecureActor = accepted.SecureActor;
    _remoteSession = accepted.Session;
    _remoteSessionHandshakeActor = Sender;
    _currentState = HandshakeState.Accepted;
   _sessionManager.StartSecureSessionOutbound(new OutboundCreateSecureSession(
            accepted.SharedSecret, accepted.SecureActor, accepted.SessionActor, accepted.Ses
});

Receive<OutboundSecureSessionInformation>(outboundSecure =>
{
    _localSecureActor = outboundSecure.LocalSecureActor;
    _remoteSessionHandshakeActor.Tell(new HandshakeChannelOpen(_localSecureActor));
    BecomeOpen();
});
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

In this sample, we see that the `SessionHandshakeActor` -when used on the OUTBOUND side of the association-is saving the `Sender` of the `HandshakeAccepted` message and then later `Tell` ing that `Sender` (whose `IActorRef` is now saved) the `HandshakeChannelOpen` message.

Use Pub/Sub

Pub/sub is delightfully simple with Akka.NET and it's a natural extension of the reactive programming style inherent to actors. Pub/sub and reply to sender are the go-to messaging patterns you should rely on.

Here's a simple example from `AkkaTime` of using pub/sub:

```
// in PresenceActor.cs
public PresenceActor()
{

    // ...
    //Authenticated sessions only
    Receive<SessionOpened>(sessionOpened => !sessionOpened.RemoteSession.Equals(ActiveSessio
    {
        var status = new UserStatus(sessionOpened.RemoteSession.UserName, AkkaTimeStatus.Act
            DateTimeOffset.Now);
        var address = sessionOpened.RemoteSessionManager.Path.Address;
        var connectivity = new UserConnectivityInfo(status, address);
        UpdateConnectivityInfoAndNotify(connectivity);
    });

    //Authenticated sessions only
    Receive<SessionClosed>(sessionClosed => !sessionClosed.RemoteSession.Equals(ActiveSessio
    {
        var status = new UserStatus(sessionClosed.RemoteSession.UserName, AkkaTimeStatus.Off
            DateTimeOffset.Now);
        var connectivity = new UserConnectivityInfo(status, UserConnectivityInfo.OfflineAddr
        UpdateConnectivityInfoAndNotify(connectivity);
    });
    // ...
}


private void UpdateConnectivityInfoAndNotify(UserConnectivityInfo info)
{
    var userName = info.Status.UserName;
    if (!_statuses.ContainsKey(userName))
    {
        _statuses[userName] = info;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            NotifySubscribers(userName, info);
        }
        else if(_statuses[userName].WillUpdate(info))
        {
            var oldStatus = _statuses[userName];
            var newStatus = oldStatus.Merge(info);
            _statuses[userName] = newStatus;
            NotifySubscribers(userName, newStatus);
        }
    }

    private void NotifySubscribers(string userName, object message)
    {
        if (_subscribers.ContainsKey(userName))
        {
            foreach(var sub in _subscribers[userName])
                sub.Tell(message);
        }
    }
```

Here we can see that the `PresenceActor`-the single source of truth on `AkkaTime.Server` for who is online and who isn't-is both a subscriber (of `SessionOpened` / `SessionClosed` messages) and a publisher to many remote `FriendsUIActor`s to alert them when sessions are starting and stopping.

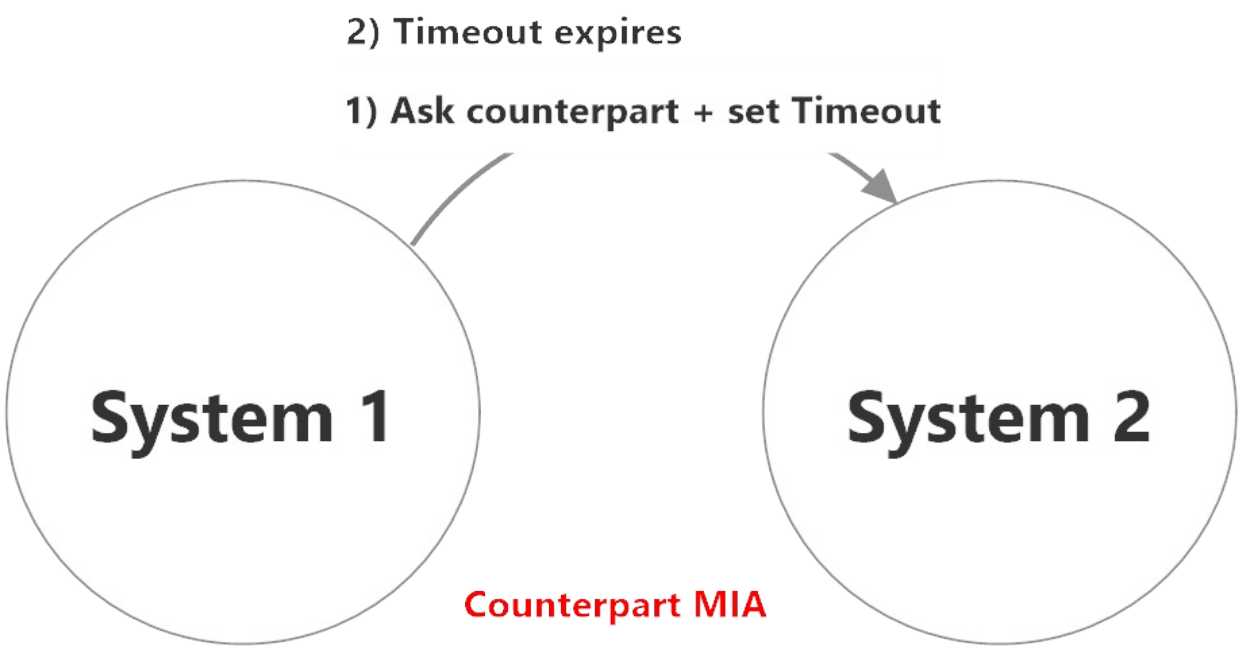
Verify Counterparts in Remote Lookup

One of the questions that comes up when dealing with communications between remote processes is this: "how do I know if anyone is listening on the other end? What if the other machine died?"

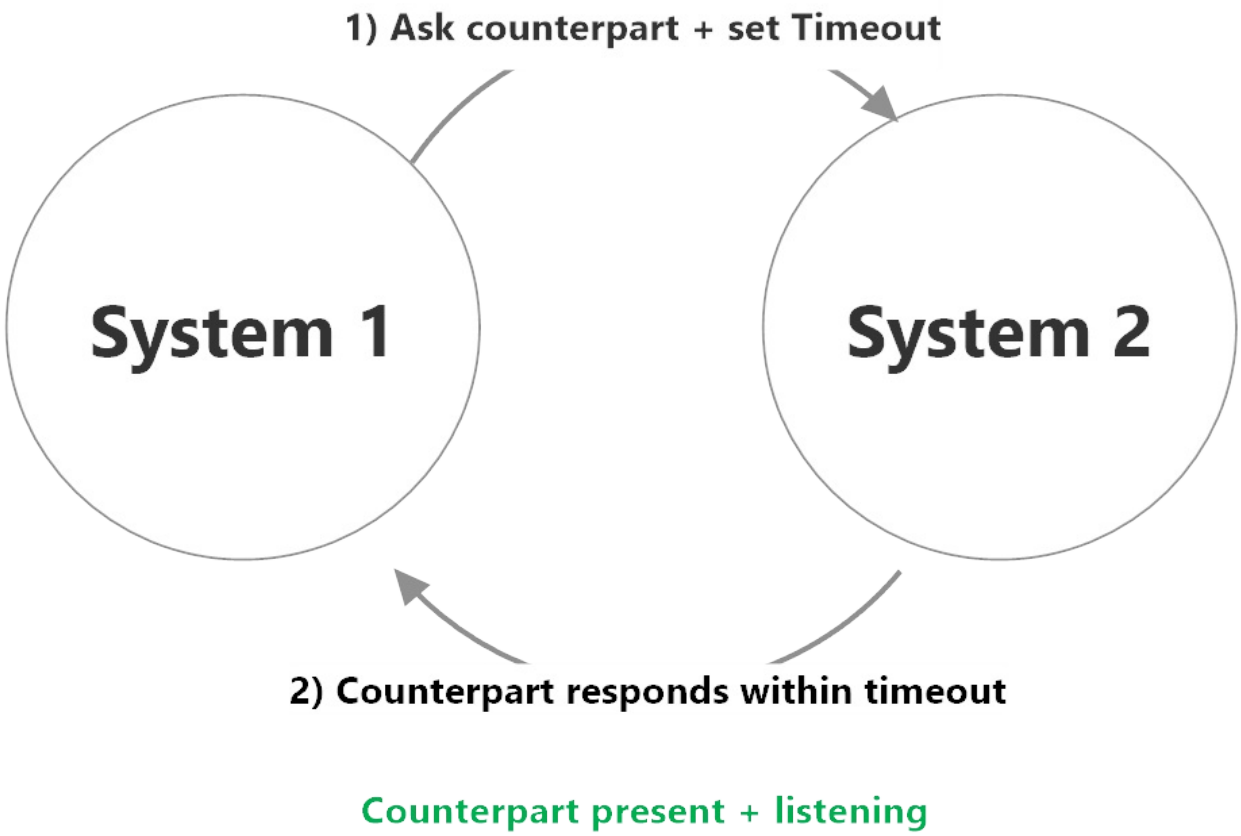
What will help is for you to verify if the other side of the remote lookup is present by making your initial message an `Ask` and setting a timeout on the `Task` it returns. If the `Task` succeeds, all is well and you can carry on. If the `Task` times out, it is treated as a negative acknowledgment and you can handle as needed.

Here's what it looks like in a failure case:

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



And in the success case:



Remotely Deploy Hierarchies with the Fan Out Pattern

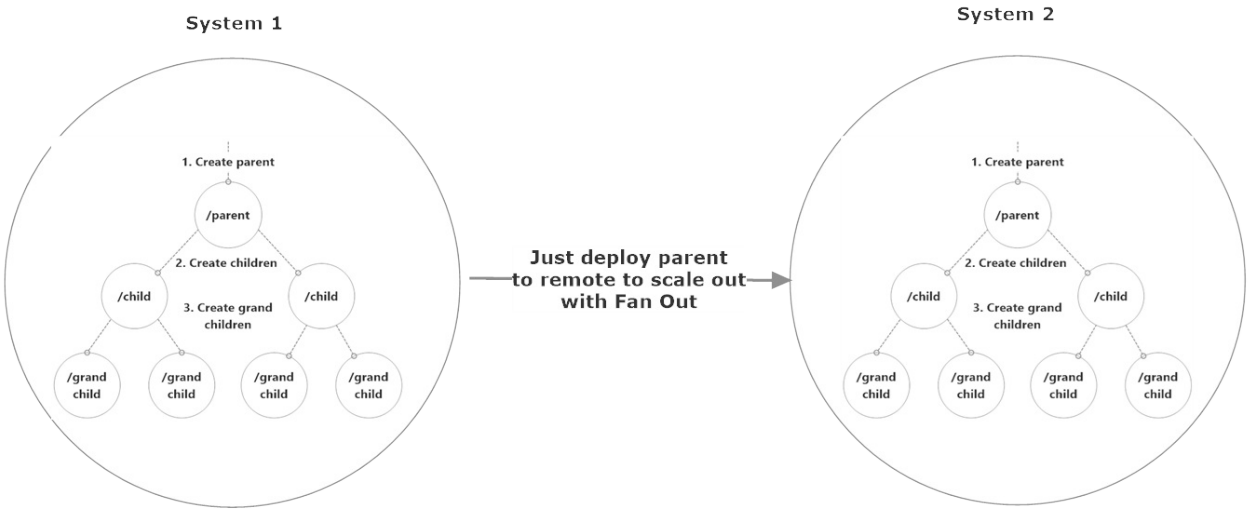
The Fan-out Pattern (one of many excellent patterns taught in our Akka.NET Design Patterns course!) is used when entire hierarchies of actors need to be able to be deployed in order to fulfill a task. Rather than have one actor responsible for creating a

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

large number of children, have a single actor be deployed who is then responsible for creating a small number of children who create grandchildren, and so on.

Essentially, you can remote deploy one actor (the parent), and then the work-processing hierarchy underneath that parent can unpack itself.

Here's what that looks like:



Next Steps

Now that we've understood how to design our actor hierarchies with Akka.Remote in the picture, let's learn a bit about how to secure our communications in the context of Akka.Remote.

In the next section we'll talk about Secure Communications in Akka.Remote.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Secure Communication in Akka.Remote

One question that comes up frequently with Akka.Remote is the story around security, and we will explore that in this section.

Built-in Security in Akka.Remote

This section is extremely brief - as of today there are no built-in secure transports in Akka.Remote. Security is a "do it yourself" exercise.

That being said, however, Helios - the socket library that powers all of the transports in Akka.Remote, is working on plans to introduce TLS and DTLS inside Helios 2.0, which will bring asymmetric encryption and secure sockets to Akka.Remote.

But in the meantime, we have to create our own security options around Akka.Remote.

Application-Specific Security Techniques

Here in this section we will discuss a small number of techniques you can use to enforce different types of security functions within your Akka.Remote applications, including:

1. Private Networks and VPNs and
2. Restricting Messages to Local-only Senders.

> NOTE: The AkkaTime sample also includes examples of how to implement two-way symmetric encryption over the network, but it's complex and beyond the scope of this course.

Secure the Network: Akka.Remote with Virtual Private Networks

The absolute best practice for securing remote Akka.NET applications today is to make the network around the applications secure - don't use public, open networks! Instead, use a private network to restrict machines that can contact Akka.Remote processes to ones who have your VPN credentials.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Some options for doing this:

- OpenVPN - for "do it yourself" environments;
- Azure Virtual Networks - for Windows Azure customers; and
- Amazon Virtual Private Cloud (VPC) - for Amazon Web Services customers.

Restrict Messages to Local Senders

One thing you can do to provide a limited level of security within some parts of your Akka.Remote application is to restrict message deliveries such that they're from locally-deployed senders only.

We have a convenient extension method built into AkkaTime which does this.

```
/// <summary>
/// Used for checking the locality of key messages
/// </summary>
public static class SecureActorRefExtensions
{
    public static bool IsLocal(this IActorRef actor)
    {
        return actor.IsNobody() || ((IActorRefScope) actor).IsLocal;
    }
}
```

We can use this extension method to check the `Scope` of an `IActorRef` and ensure that it's a local sender.

We use this extensively inside `AkkaTime.Server` to ensure that authentication messages can't be spoofed by clients.

```
/* Inside AkkaTime.Server.ServerSessionManager */
Receive<AuthenticationSuccess>(success =>
{
/* BOYEAH - THINGS ARE HAPPENING */
if (Sender.IsLocal())
{
    var authedSession = new ActiveSession(success.User.DisplayName, SessionIdGenerator(),
        DateTimeOffset.Now + DefaultSessionLength);
    _remoteSession = authedSession;
    _securePublisher.Tell(new SecurePublish(RemoteSessionActor, Self, new CreateSession(_rem
    BecomeAuthorized();
}
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
else //if the sender wasn't local, then we can't trust the results of the message
{
    _securePublisher.Tell(new SecurePublish(RemoteSessionActor, Self,
    new AuthenticationFailure(success.User.DisplayName,
    "Authentication results can only come from local senders.")));
}
});
```

We also used the Parent Proxy Pattern from our *Akka.NET Design Patterns* course to create the `LocalOnlyForwarder`, an invisible router that will only forward a message onto the downstream actor if the sender is local.

```
/// <summary>
/// Transparenty proxy that only forwards messages on from local senders.
/// </summary>
public class LocalOnlyForwarder : UntypedActor
{
    //Props that belong to the actor we're forwarding to
    private readonly Props _childProps;
    private Action<object, IActorRef> _nonRemoteMessageHandler;

    private IActorRef _destination;
    public static readonly Action<object, IActorRef> DefaultUnsupportedMessageHandler = (o,

    public LocalOnlyForwarder(Props childProps, Action<object, IActorRef> nonRemoteMessageHa
    {
        _childProps = childProps;
        _nonRemoteMessageHandler = nonRemoteMessageHandler ?? DefaultUnsupportedMessageHandl
    }

    protected override void PreStart()
    {
        _destination = Context.ActorOf(_childProps);
    }

    protected override SupervisorStrategy SupervisorStrategy()
    {
        // ESCALATE all failures back to the parent
        return new OneForOneStrategy(new DeployableDecider(Directive.Escalate));
    }

    protected override void OnReceive(object message)
    {
        if (Sender.IsLocal())
        {
            _destination.Forward(message);
        }
        else
        {
            DefaultUnsupportedMessageHandler(message, Sender);
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            }
        }
    }
```

Next Steps

Now that we've talked a little bit about security in the context of Akka.Remote, let's start thinking about how we architect entire applications knowing that remote actors and associations will be involved from the start.

In the next section we'll talk about **Designing Distributed Systems with Akka.Remote**.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Distributed Systems Architecture with Akka.Remote

In this section we're going to focus on two specific types of network architectures and help you create a framework for deciding which ones you should work with, and then we'll offer some guidance on how to build each with Akka.Remote.

Picking a Network Architecture

There are really only two families of network architectures:

1. **Client-Server** and
2. **Peer-to-Peer**

Everything else is a hybrid architecture of some sort.

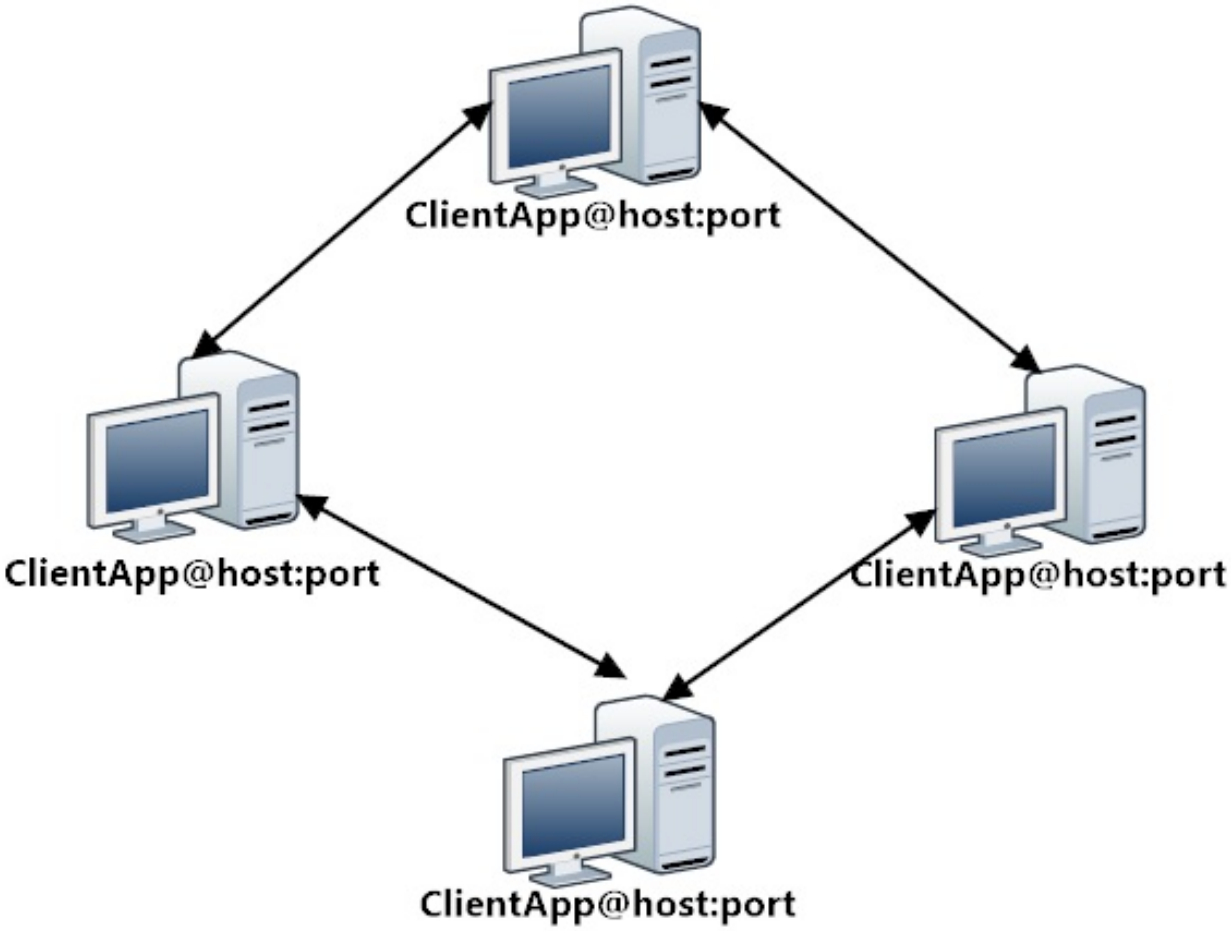
`AkkaTime` uses both architectures concurrently, which also happens regularly with popular types of social applications such as chat clients (duh!) and multiplayer video games.

So what's fundamentally different about each?

Peer-to-Peer

In peer-to-peer architectures, a single application is designed to communicate uniformly with other copies of itself exactly. In other words, each individual client is designed to work with other copies running the exact same code, but potentially with different state.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



Peer-to-peer architectures can take on a lot of different shapes. You'll notice in the diagram above that there isn't a connection between every single node. That's perfectly acceptable in many peer to peer architectures!

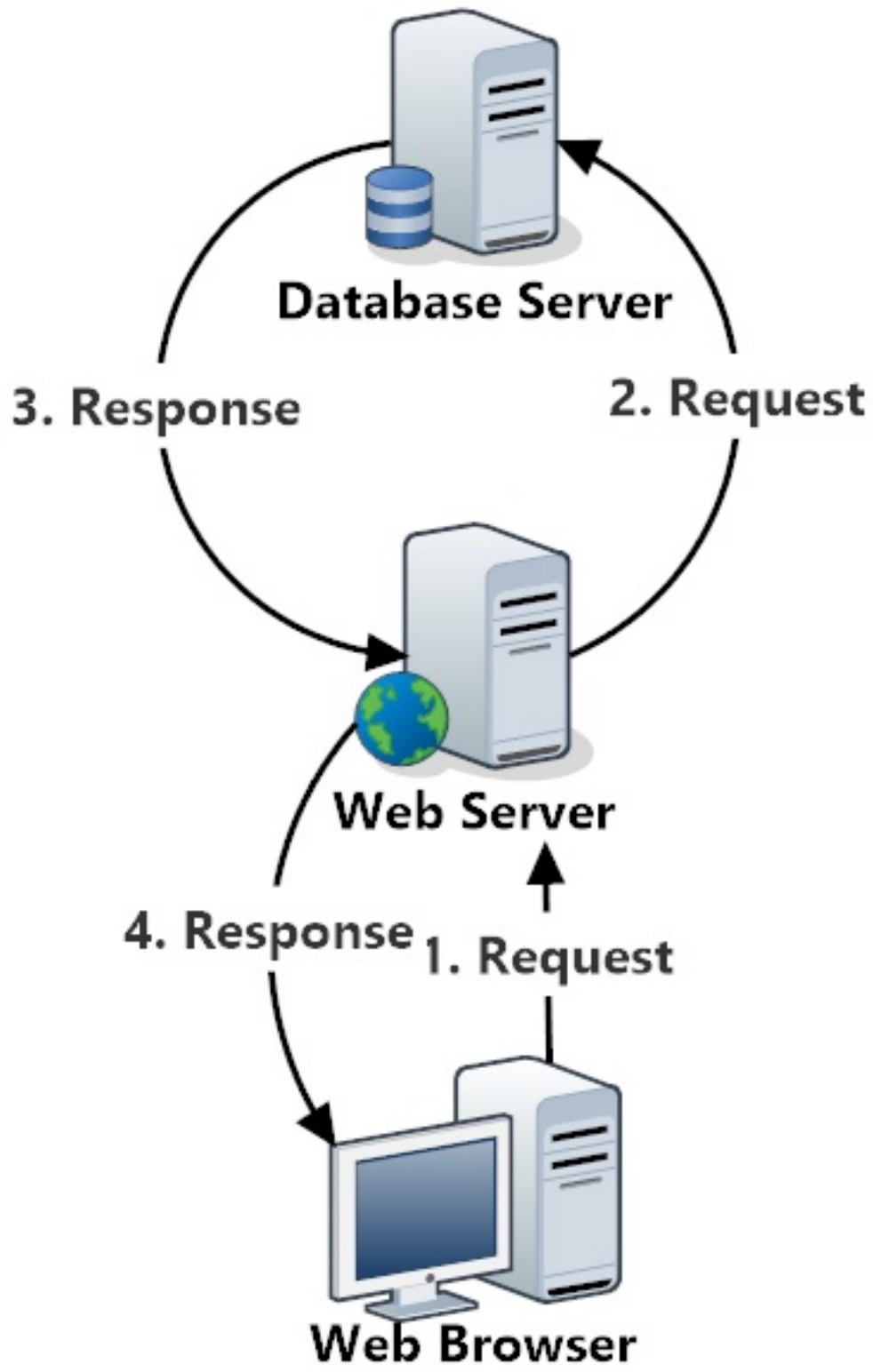
The other thing you'll notice is that information flows in both directions - everyone's equal in a peer-to-peer network. There are no one-way dependencies.

Client-Server

In client-server architectures, multiple different services are designed to fulfill specific roles that meet each other's needs. The "application" is effectively an aggregate of two or more different service implementations working together cooperatively. In a client-server relationship the server service exists independent of the client service but the client depends entirely on the server.

A service that is a client in one context can be a server in another. The "front-end" of your web application is effectively the server for the web browser, but it's also a client of your data service.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

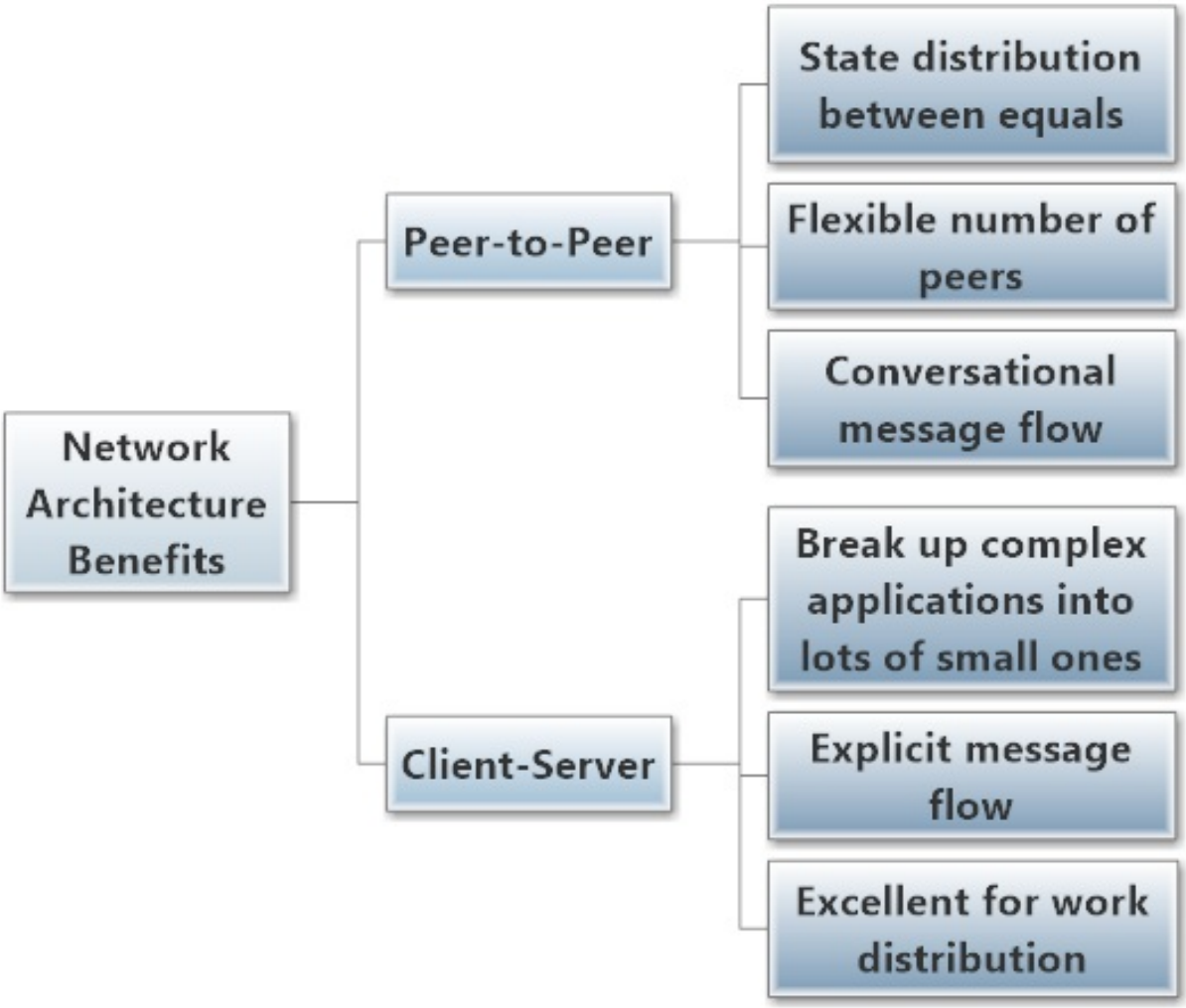


In a client-server architecture, clients depend on information flowing from the server. Like peer-to-peer architectures there can be bidirectional communication, although it's usually in the form of requests from the client and responses from the server.

When Should You Use Each?

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

This is a trick question. **In high availability scenarios, you'll want to use both**.



Each architecture brings with it different benefits. The way we break it down is thus: how do you want to mutate state inside your application?

If you have *a lot* of state that you have to track concurrently, like more than can economically fit in memory or on disk for a single machine, then you need to use a peer-to-peer architecture for state distribution. This is how distributed databases like Cassandra work.

Or let's say you have a conversational message flow that might be extremely chatty at times, such as a multiplayer video game or chat service. You shouldn't let a server be a single point of failure for that scenario. Use peer-to-peer connectivity to provide resiliency against any individual network device failure killing off your application.

But what if you have a really complex business domain with hundreds of different things you need to do in order to serve your customers? Then you need to break up your application into distinct services with client-server relationships.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Or what if you have a complex, but ultimately linear workflow for processing large ETL workloads? You're probably better off breaking up the steps of that job into distinct services and making them clients of each other, since the message flow is explicit with a clear direction.

You can use peer-to-peer architectures and client-server architectures simultaneously. But just make explicit choices about each.

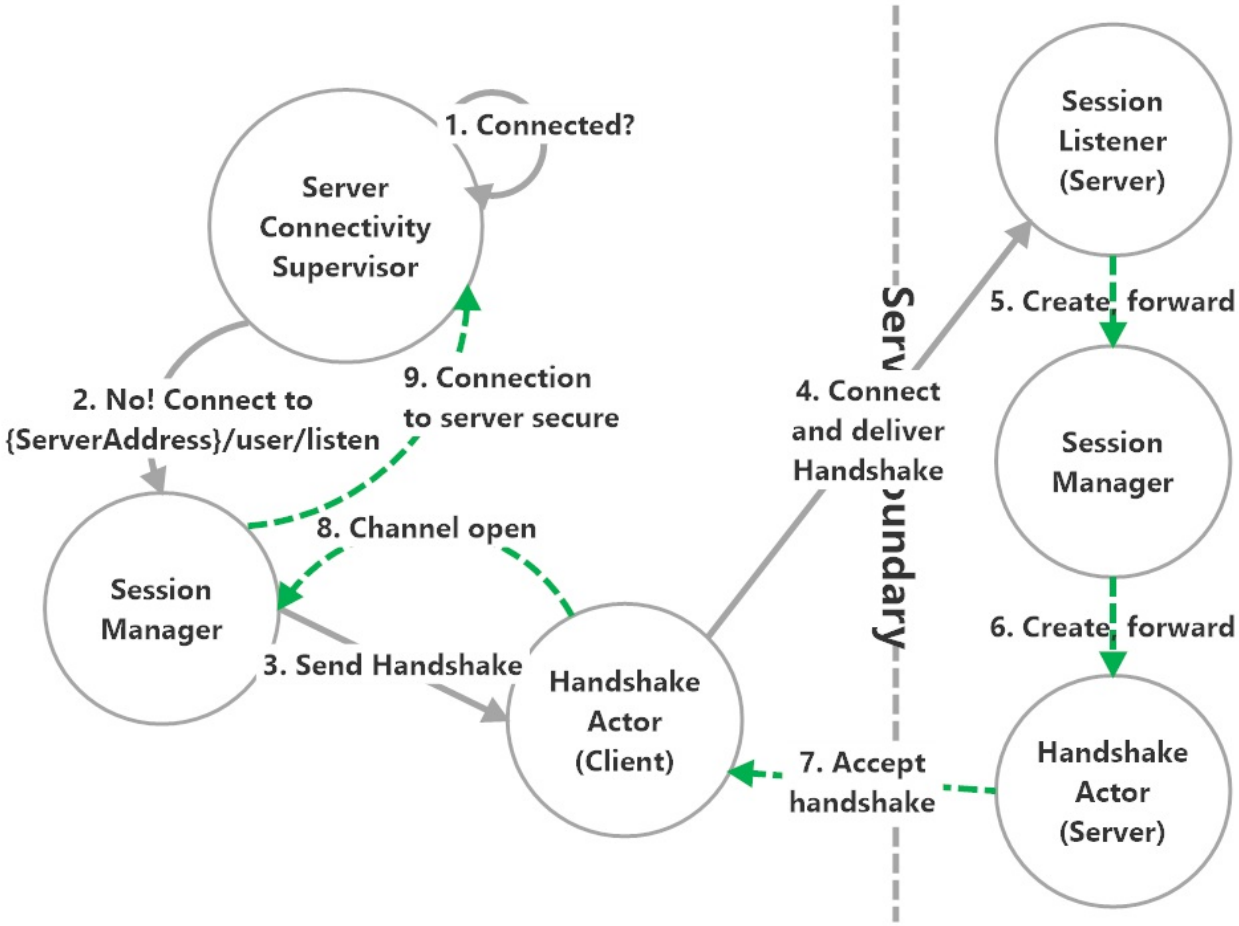
Client-Server Architectures with Akka.Remote

In order to build a basic client-server architecture with Akka.Remote, you have to have three ingredients:

1. Clients need some way of contacting the server;
2. Server and Client need to share common message definitions; and
3. Client and Server need to have some initial message contracts / expected message flows in order to propagate `IActorRef`s to the rest of their actors.

Here's what this looks like inside AkkaTime!

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



So when the `AkkaTime.Client` boots, it loads the following piece of information from its HOCON configuration inside App.config:

```
akkatime{
 sessions{
  handshake-timeout = 4.0s #max time to complete the handshake timeout process
  verify-timeout = 1.0s #max time to authentication an end-user locally on the server
  verify-chat = on
 }
 server = "akka.tcp://akkatime@127.0.0.1:14445"
}

akka {
    #rest of the akka configuration
}
```

The `akkatime.server` value is read during the `AkkaTime.Client` startup:

```
var actorSystem = ActorSystemRefs.AkkaTimeSystem = ActorSystem.Create("akkatime");
var serverAddress = ActorSystemRefs.ServerAddress = Address.Parse(actorSystem.Settings.Confi
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

And passed to the `ServerConnectionSupervisor` as a constructor argument:

```
public ServerConnectionSupervisor(IActorRef serverConnectivityStatus, Address serverAddress,
{
    _serverConnectivityStatus = serverConnectivityStatus;
    _serverAddress = serverAddress;
    _cryptKey = cryptKey;

    _serverConnectivityStatus.Tell(new SubscribeServerConnectivity(Self));
    Self.Tell(new ConnectToServer());
    Disconnected();
}
```

The `ServerConnectionSupervisor` immediately receives a `ConnectToServer` message upon startup, which it uses to bootstrap its `ClientToServerSessionManager` and `SessionHandshakeActor` in order to open remote communication with the `AkkaTime.Server` instance.

```
private void Disconnected()
{
    Receive<ConnectToServer>(server =>
    {
        var session = ActiveSession.AnonymousSession;
        _sessionManager =
            ActorSystemRefs.ClientActor =
                Context.ActorOf(
                    Props.Create(
                        () => new ClientToServerSessionManager(_cryptKey, session)), ActorPa
        var sessionManagerWrapper = new SessionManagerWrapper(false, _sessionManager);
        var handshakeActor =
            Context.ActorOf(SessionHandshakeActor.OutboundProps(sessionManagerWrapper, sessi
                (Context.System.Settings.Config.GetTimeSpan("akkatime.sessions.handshake-tim
                    TimeSpan.FromSeconds(4.0)))));

        //subscribe the connectivity actor to all lifecycle events, including debug events
        _sessionManager.Tell(new SubscribeToSessionLifecycleEvents(_serverConnectivityStatus

        var remoteEndpoint =
            Context.ActorSelection(new RootActorPath(_serverAddress) / "user" / ActorPaths.S
        var connectTo = new ConnectTo(remoteEndpoint, BeginHandshake.Instance);

        _serverConnectivityStatus.Tell(connectTo);
        handshakeActor.Tell(connectTo);
        BecomeConnecting();
    });
}
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

The key line is here:

```
var remoteEndpoint =
            Context.ActorSelection(new RootActorPath(_serverAddress) / "user" / ActorPaths.S
```

In this line we use the server's `Address`, which we parsed from configuration, along with the known location of the server's `SessionListener` actor on the `AkkaTime.Server` actor hierarchy (`/user/listen` in this case, if we care to look inside the `ActorPaths` class inside the `AkkaTime.Client` project.)

We combine both of these pieces of information to create an `ActorSelection` that resolves to the following full `ActorPath` in this example: `akka.tcp://akkatime@127.0.0.1:14445/user/listen`.

When the `AkkaTime.Server` is running with its default configuration, the one the AkkaTime code sample ships with, its `SessionListener` will be reachable at this `ActorPath` and remote communication between the client and server will be successful.

From there other types of communication between client and server will open up as a result of the two session manager actor implementations on either end of the association, but this is all that's needed to bootstrap a client-server relationship.

All of the message types are already shared, and even the actor types used to initiate the remote commnication process are both shared between client and server, so the expected message types and message flows will match. We even unit test this initial flow inside the `AkkaTime.Common.SecureActors.Tests` project.

Peer-to-Peer Architectures with Akka.Remote

Peer-to-Peer architectures require *all of the same things* that a Client-Server architecture does, with a couple of additional tidbits:

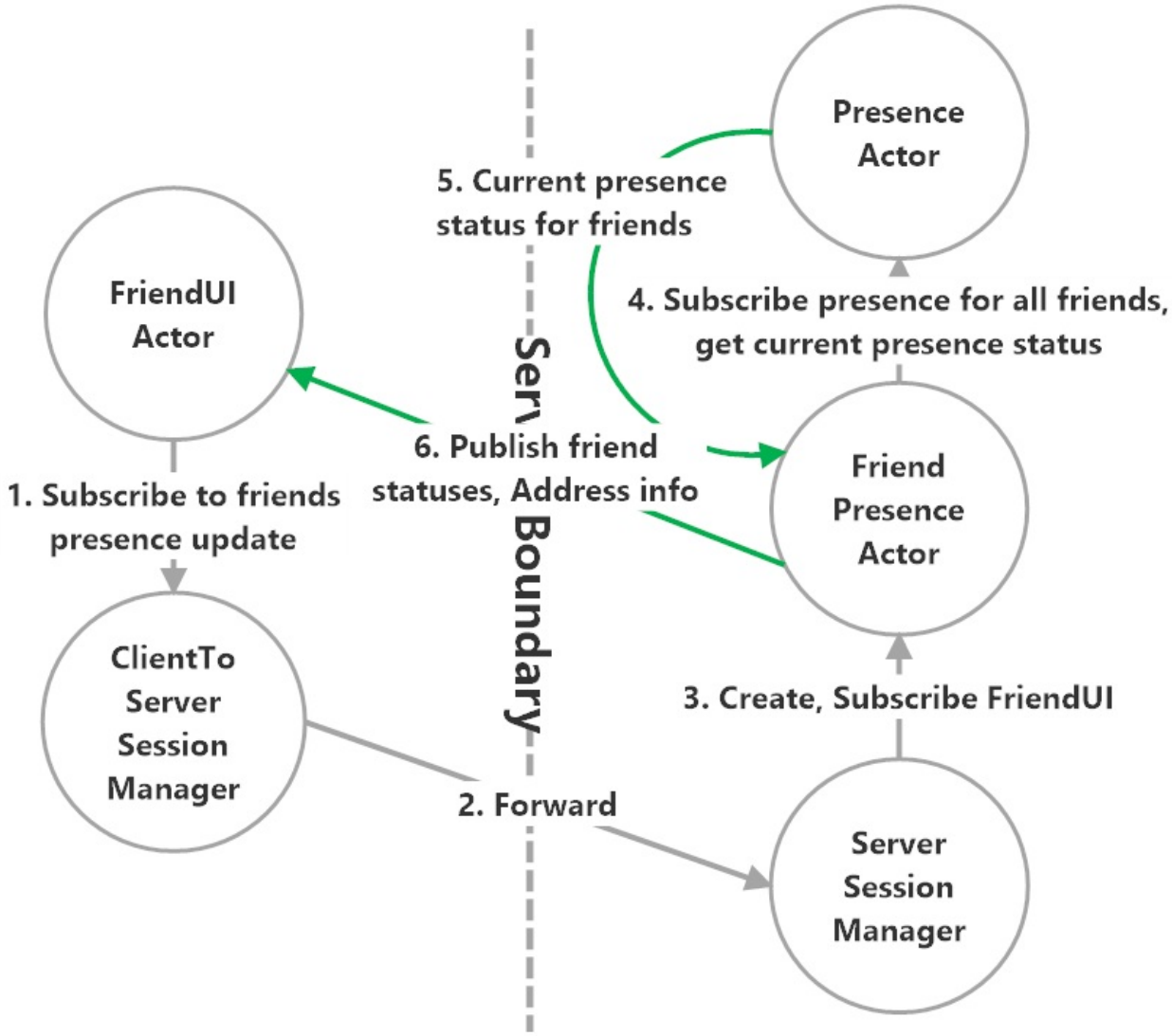
1. Peers need a "peer list" - a list of available peers to connect to and

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

2. Peers need the ability to initially hydrate their peer list when they first start.

Here's the great irony of peer-to-peer architectures: **seeding a peer list usually requires some form of client-server communication**. Argh! We just can't escape client-server!

So after an `AkkaTime.Client` instance has successfully authenticated with an `AkkaTime.Server`, here's what the process for fetching an initial seed list looks like:



So for all intents and purposes, here's what the master peer list looks like inside the `PresenceActor`, the single source of truth for all presence information. It contains a `Dictionary<string, UserConnectivityInfo>` where the key corresponds to the name of a verified, registered user.

```
/// <summary>
/// A user has been seen active at a particular address, and here's the information
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
/// used for tracking that data.
/// </summary>
public class UserConnectivityInfo : IPresenceEvent
{
    public UserConnectivityInfo(UserStatus status, Address address)
    {
        Address = address;
        Status = status;
    }

    public UserStatus Status { get; private set; }

    /// <summary>
    /// The remote address we can use to contact the user
    /// </summary>
    public Address Address { get; private set; }

    public bool WillUpdate(UserConnectivityInfo connectivityInfo)
    {
        if (connectivityInfo == null) return false;
        return Status.WillUpdate(connectivityInfo.Status) && !Address.Equals(connectivityInf
    }

    public UserConnectivityInfo Merge(UserConnectivityInfo connectivityInfo)
    {
        if(connectivityInfo == null || connectivityInfo.Address == null)
            throw new ArgumentNullException("connectivityInfo", "connectivityInfo and addres
        return new UserConnectivityInfo(Status.Merge(connectivityInfo.Status),
            connectivityInfo.Address);
    }

    public UserConnectivityInfo Merge(UserStatus statusUpdate)
    {
        return new UserConnectivityInfo(Status.Merge(statusUpdate), Address);
    }
}
```

The `UserConnectivityInfo` class also contains some other equality members, but we've omitted them for brevity. This class represents the data that comprises the peer list for an `AkkaTime.Client` - a status and a reachable `Address` we can use to connect to online `AkkaTime` users.

This information gets published by the `PresenceActor` to `FriendsPresenceActor` on the server, which then promptly publishes this information to the `FriendsUIActor` on the client. Multiple layers of pub-sub makes all of the complexity in managing this actually pretty manageable.

```
public FriendsUIActor(FriendsForm friendsForm, IActorRef clientActorRef, IActorRef chatActor
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
{
    _friendsForm = friendsForm;
    _clientActorRef = clientActorRef;
    _chatActorRef = chatActorRef;

    //Removed other Receive methods for brevity

    Receive<UserConnectivityInfo>(connectivityInfo =>
    {
        if (_friends.ContainsKey(connectivityInfo.Status.UserName))
        {
            var friend = _friends[connectivityInfo.Status.UserName];
            friend.SetConnectivity(connectivityInfo);
        }
        else
        {
            AddFriend(new Friend(connectivityInfo.Status.UserName), connectivityInfo);
        }
    });
}
```

When the `FriendsUIActor` receives a local notification about a change in presence, it updates a local copy of its peer list and updates the UI to let the end-user know if any of his or her friends are available for chat.

Next Steps

Now that you've learned how easy it is to implement universal network architectures with Akka.Remote, let's explore some patterns for dealing with the harsh realities of imperfect networks. Let's move onto **The Realities of the Network: Latency, Failures, and Dead Processes**.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Working with the Realities of the Network

Programmers often fall victim to the fallacies of distributed computing when building networked applications, so in this section we want to spend some time address both the realities of network programming AND how to deal with those realities powerfully with Akka.Remote.

The Realities of the Network

The biggest thing we have to confront about network programming is this:

> **The network is imperfect. Latency, dropped packets, and disconnects are inevitable consequences of building distributed software.**

In particular, we're going to focus on three types of network issues that we can deal with powerfully via tools that are built into Akka.Remote:

1. Latency
2. Network failures and dropped messages; and
3. Dead or hung processes.

Rather than write distributed software and "hope for the best" when it comes to the reliability of the network, let's take some active measures to protect the consistency and availability of our applications by anticipating and actively planning for these types of failures to happen.

Latency Mitigation Techniques with Akka.Remote

When we talk about "latency," here's what we mean:

> Latency is the time interval between a cause and an effect of some change in a networked system. In HTTP terms, a cause might be a request and an effect

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

> might be a response.

Latency is a problem for operations which have real-time computing requirements, and in Akka.NET applications it's pretty common to have soft real-time requirements.

So when latency increases as a result of network congestion or a decrease in responsiveness from a participating remote process, this can degrade the quality of service or even the availability of a service if not dealt with powerfully.

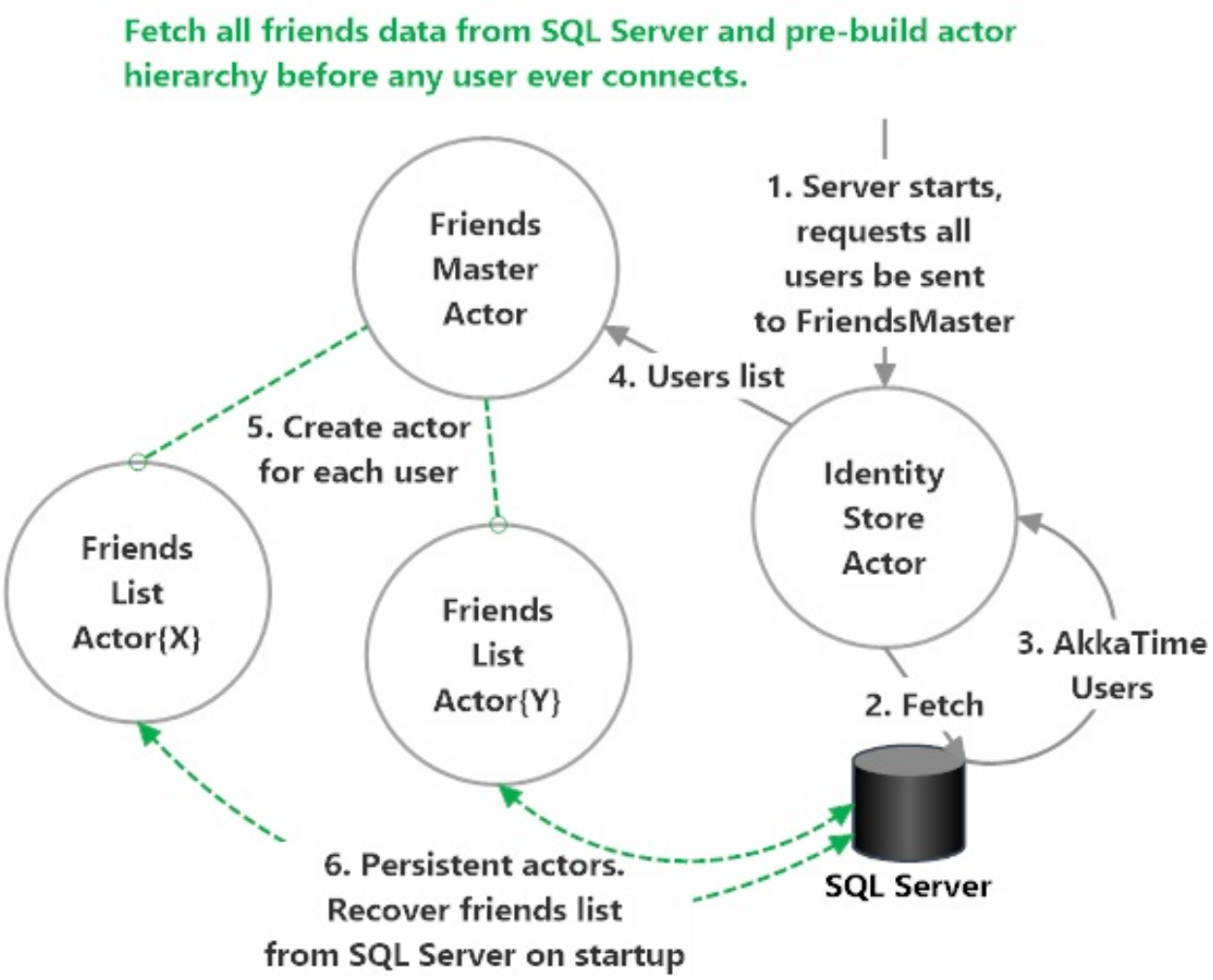
In this section we're going to explore three techniques for mitigating the effects of latency on your system's responsiveness:

1. **Pre-fetch** - fetching and caching network resources before they're needed in order to avoid run-time latency;
2. **Publish-subscribe (pub-sub)** - a messaging pattern that delivers network state literally as fast as possible; and
3. **Timeout** - a technique designed to terminate a long-running operation past some point of tolerance in order to provide high responsiveness and quality of service to other requests.

Pre-fetch Technique

The Pre-fetch technique has a couple of other names you might be familiar with already, such as "cache warming," but the idea is pretty straightforward.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



In the AkkaTime example above, we populate the friends lists for all registered AkkaTime users and create the actor hierarchy for managing future friends requests ahead of time. This "front-loads" all of the latency that would arise when a user first logs in and has it happen when the server starts up instead.

The idea is to **fetch network resources you'll need to fulfill requests before any requests are ever made**. This might include fetching a bunch of data from a database and caching it in-memory on each server before the server begins accepting requests, so it can use those data objects already in-memory instead of having to make a second network call to fetch them from somewhere else.

So let's take a look at the code for doing this inside AkkaTime:

```
/* Inside AkkaTime.Server.AkkaTimeService.cs */
AkkaTimeSystem = ActorSystem.Create("akkatime");
var resolver = NinjectResolver = new NinjectDependencyResolver(kernel, AkkaTimeSystem);

// Use DI with Ninject to get new SqlConnection objects
var identityProps = resolver.Create<IdentityStoreActor>();
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
var identity =
    AkkaTimeSystem.ActorOf(
        Props.Create(() => new LocalOnlyForwarder(identityProps, null)).WithRouter(FromConfi
        ActorPaths.Identity.Name);

var presence =
    AkkaTimeSystem.ActorOf(
        Props.Create(() => new LocalOnlyForwarder(Props.Create<PresenceActor>(), null)), Act

var presenceSubscriberProps = Props.Create(() => new PresenceSubscriber(presence));

var friendsMaster = AkkaTimeSystem.ActorOf(Props.Create(() =>
    new FriendsMaster(presence, identity, FriendListPropsFactory)), ActorPaths.Friends.Name)

//pre-warm the friends list actors
identity.Tell(new IdentityStoreActor.FetchAllUsers(), friendsMaster);
```

In the last line of this snippet from `AkkaTime.Server.AkkaTimeService.cs` you can see that we tell the `/identity` actor to `FetchAllUsers` and that the sender of this message is `/friends`, the `FriendsMaster`.

```
/// <summary>
/// Actor reponsible for fetching users from the local system
/// </summary>
public class IdentityStoreActor : ReceiveActor
{
    #region Message classes


    /// <summary>
    /// Fetch all of the users stored in the database. Typically used
    /// only on pre-fetch
    /// </summary>
    public class FetchAllUsers { }

    #endregion

    private readonly Func<IUserRepository> _userRepoFactory;
    private ILoggingAdapter _log = Context.GetLogger();

    public IdentityStoreActor(Func<IUserRepository> userRepoFactory)
    {
        _userRepoFactory = userRepoFactory;

        /* Removed other Receive methods for brevity */

        Receive<FetchAllUsers>(users =>
        {
            var repo = _userRepoFactory();
            var sender = Sender;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            repo.FetchAllUsers().ContinueWith<object>(tr =>
            {
                repo.Dispose();
                return tr.Result.Payload;
            }).PipeTo(sender);
        });
    }
}
```

The `IdentityStoreActor` receives the `FetchAllUsers` message and loads all of the the AkkaTime users from SQL Server, and then pipes the result back to the "sender" - which in this case is actually the `FriendsMaster`.

```
/// <summary>
/// Master actor responsible for the management of friends lists
/// </summary>
public class FriendsMaster : ReceiveActor
{
    private readonly IActorRef _presenceActor;
    private readonly IActorRef _identityActor;
    private readonly Func<string, Props> _friendListPropsFactory;

    public FriendsMaster(IActorRef presenceActor,  IActorRef identityActor, Func<string, Pro
    {
        _presenceActor = presenceActor;
        _friendListPropsFactory = friendListPropsFactory;
        _identityActor = identityActor;

        // Warm up all of the persistent friends list actors
        // and warm the presence actor
        Receive<IEnumerable<ExistingUser>>(users =>
        {
            foreach (var user in users)
            {
                CreateIfNotExist(Context, user.DisplayName);

                //warm up the presence actor
                _presenceActor.Tell(new PresenceActor.InitialPresence(user.DisplayName));
            }
        });

       /* Removed other Receive methods for brevity */
    }

    private IActorRef CreateIfNotExist(IActorContext context, string userName)
    {
        var child = context.Child(userName);
        if (child.Equals(ActorRefs.Nobody))
        {
            return context.ActorOf(_friendListPropsFactory(userName), userName);
        }
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            return child;
        }
    }
```

The `FriendsMaster` actor, in turn, iterates through the list of results delivered to it from the `IdentityStoreActor` and creates a `FriendsListActor` instance for each `ExistingUser`.

The `FriendsListActor` is a persistent actor and will automatically build the friends list for that particular user by replaying all of the `AddFriend` and `RemoveFriend` commands it's ever received from the journal stored in SQL Server.

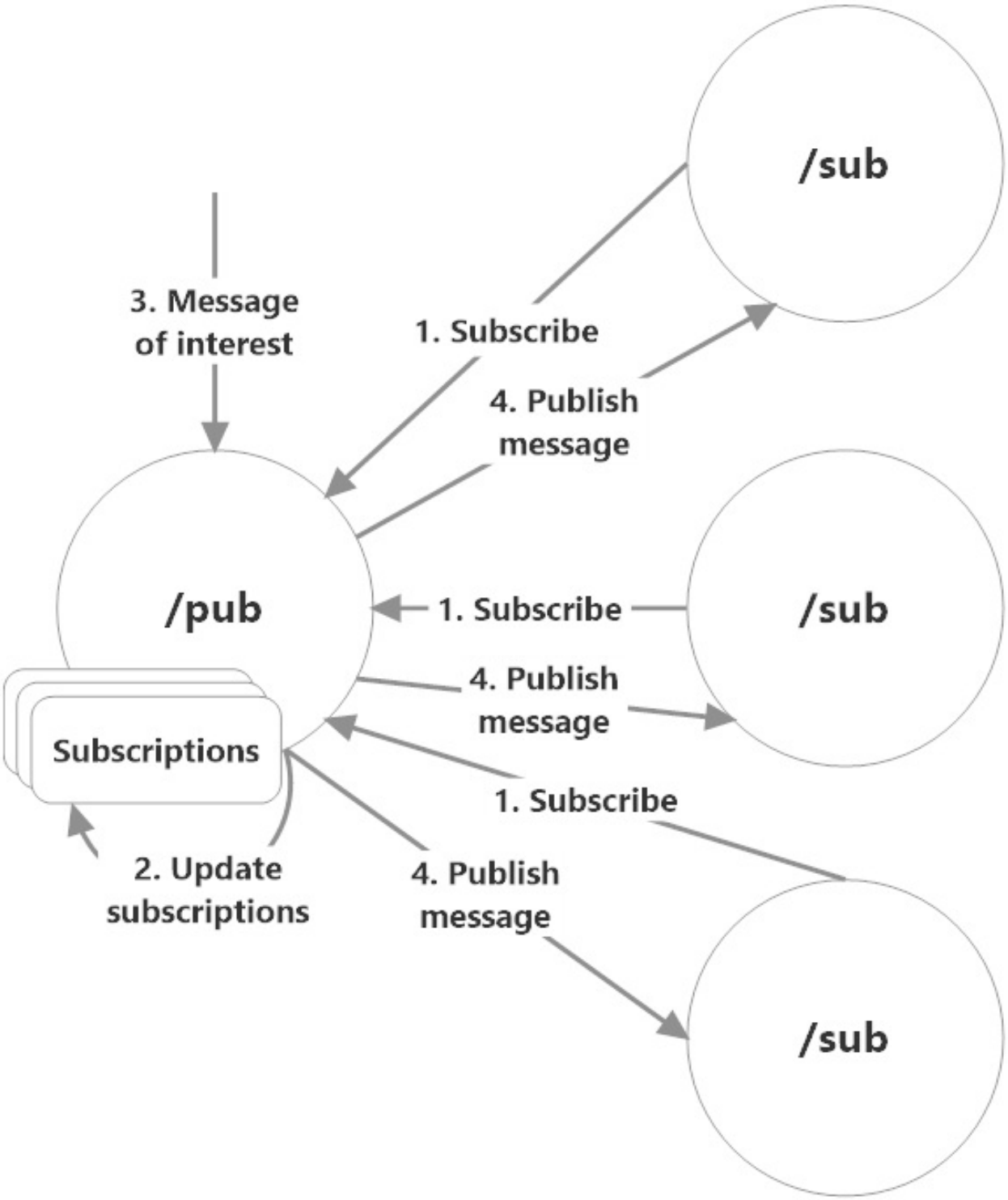
We do all of this work **before we ever get any requests from `AkkaTime.Client`**, so the `AkkaTime.Server` is able to serve friend lists to clients when they start at drastically reduced latency and load times.

Publish-subscribe (pub-sub) Messaging Pattern as a Latency Reduction Technique

In our *Akka.NET Design Patterns* course we go into some detail on the Pub-sub messaging pattern and its potential use cases.

One of the massive benefits of this messaging pattern is that it eliminates latency as a concern by flipping our normal "request / response" way of thinking about the network on its head.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



In pub-sub, a requestor sends a "subscribe" message to a publish and then receives periodic updates related to that subscription in perpetuity or until it unsubscribes.

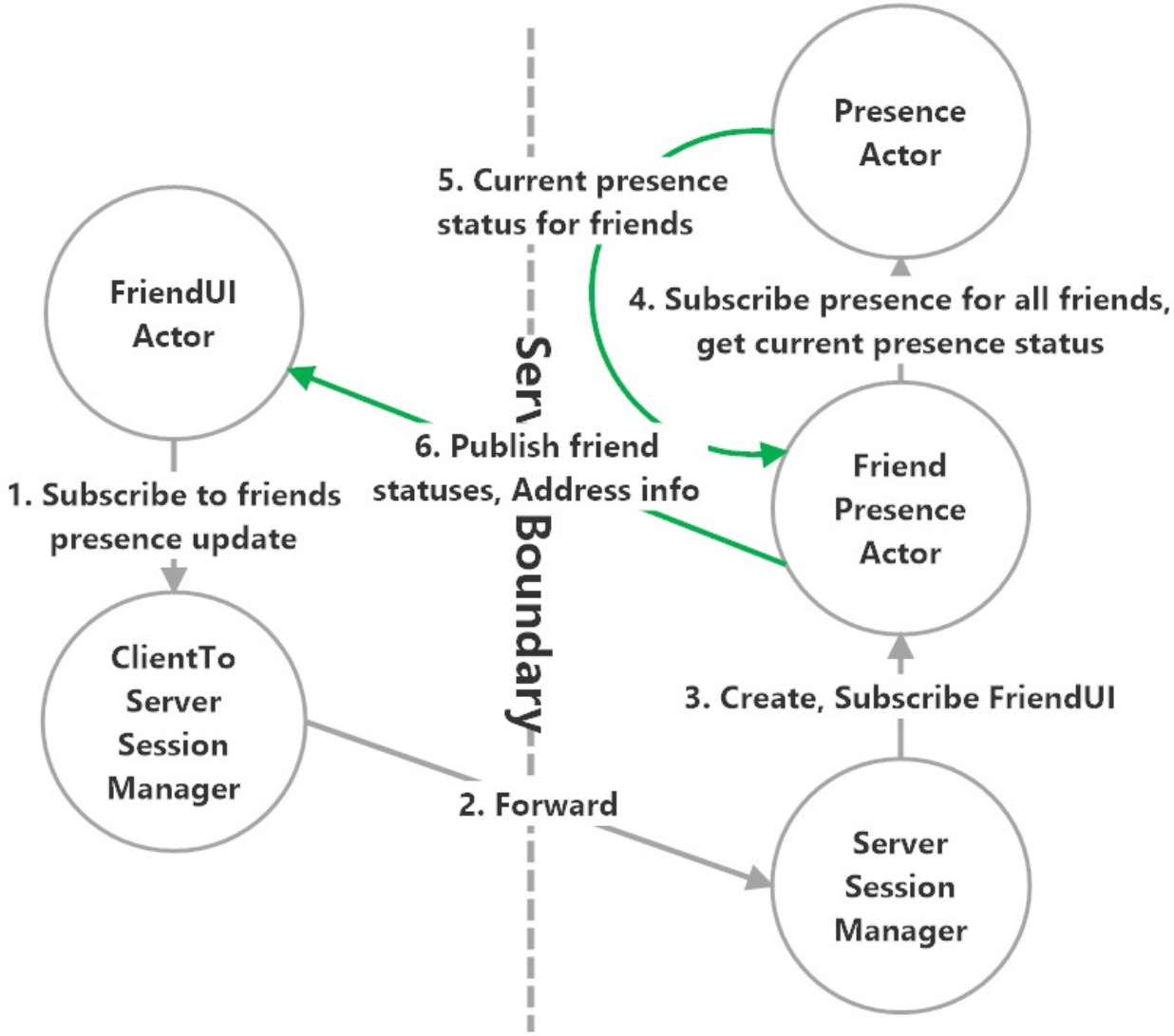
So what's a big consequence of this? **State gets pushed into your subscribing actors as fast as your application can possibily allow**.

With pub-sub you can assume that your "subscriber" actors are always operating with the latest available state - they don't need to go looking somewhere else to find it. And

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

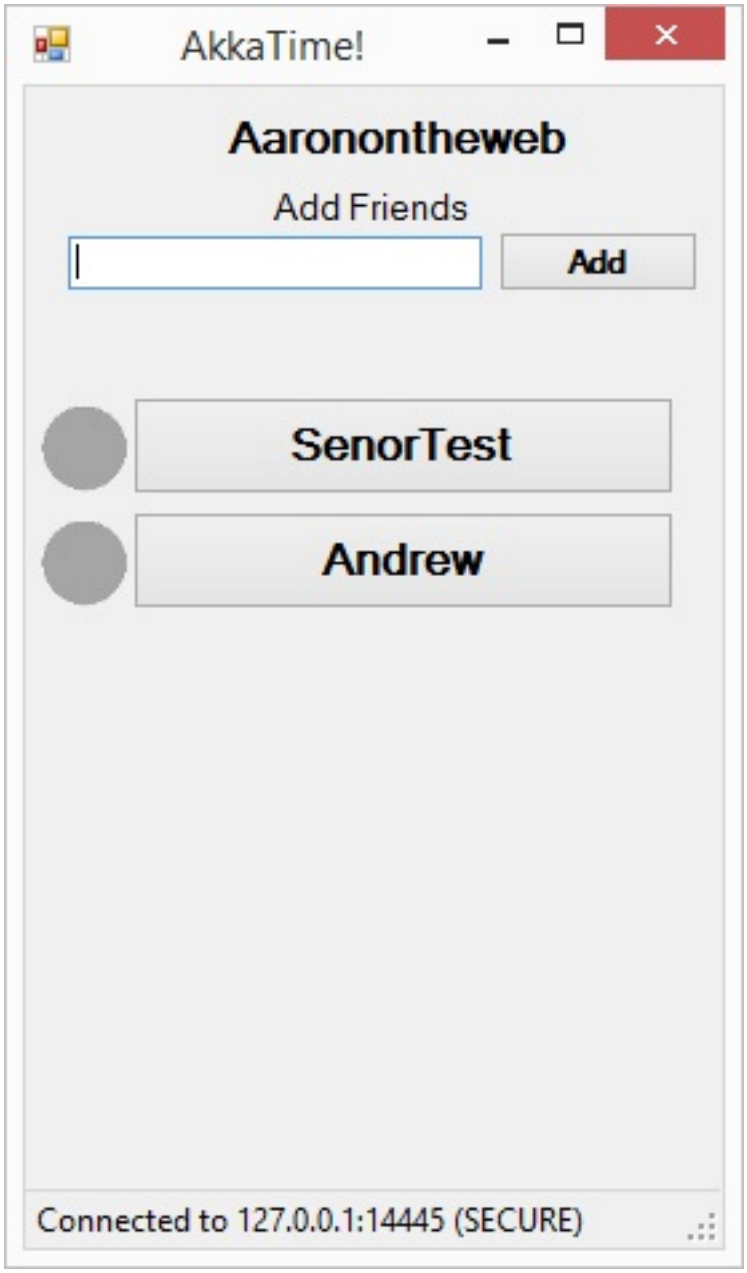
they get notified as soon as it changes! This is one of the massive benefits of reactive programming - your subscriber actors just need to react to changes in state as they happen (and they get notified *as fast as possible*,) rather that try to request that state from somewhere else.

Let's take a look at an example from the AkkaTime example.



In this instance, the `FriendUIActor` - the actor that literally updates this screen inside the `AkkaTime.Client` application:

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



The `FriendUIActor` tells its local `ClientToServerSessionManager`, the actor responsible for establishing and routing secure communication between an `AkkaTime.Client` and an `AkkaTime.Server` instance, to subscribe it to updates about friends becoming available or unavailable.

Let's see what the code looks like for this.

```
/// <summary>
/// Actor responsible for managing UI updatesto the <see cref="FriendsForm"/>
/// </summary>
public class FriendsUIActor : ReceiveActor
{
    private readonly FriendsForm _friendsForm;
    private readonly IActorRef _clientActorRef;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
    private readonly IActorRef _chatActorRef;

    private readonly Dictionary<string, UserStatusLauncher> _friends = new Dictionary<string

    public FriendsUIActor(FriendsForm friendsForm, IActorRef clientActorRef, IActorRef chatA
    {
        _friendsForm = friendsForm;
        _clientActorRef = clientActorRef;
        _chatActorRef = chatActorRef;

        /* Receive methods removed for brevity */

    }


    protected override void PreStart()
    {
        // Kicks off a process that has the server start subscribing this actor
        // to changes in friend availability and presence
        _clientActorRef.Tell(new BootstrapFriendsPresence(Self));
    }
}
```

You can see that the `FriendsUIActor` sends a `BootStrapFriendsPresence` message to the `_clientActorRef`, an `IActorRef` that is a handle to the `ClientToServerSessionManager`, during its `PreStart` routine.

```
/* Authenticated behavior inside AkkaTime.Client.ClientToServerSessionManager */

 /// <summary>
/// Authenticated behaviors
/// </summary>
private void Authenticated()
{
    // This request can only come from a friends UI actor
    Receive<BootstrapFriendsPresence>(bootstrap =>
    {
        var sender = _friendUIActor = Sender;
        _securePublisher.Tell(new SecurePublish(RemoteSessionActor, sender, bootstrap));
    });

    // other Receive methods removed for brevity
}
```

The `ClientToServerSessionManager` receives this `BootstrapFriendsPresence` message and forwards it to the `AkkaTime.Server`'s corresponding `ServerSessionManager`.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
/* Authenticated behavior inside AkkaTime.Server.ServerSessionManager */

private void AuthorizedReceives()
{
    Receive<BootstrapFriendsPresence>(friends =>
    {
        if (_userPresenceManager == null)
        {
            var userName = GetRemoteSessionInformation().UserName;
            _userPresenceManager =
                Context.ActorOf(
                    Props.Create(
                        () =>
                            new FriendsPresenceActor(GetRemoteSessionInformation().UserName,
                                _presenceActor, _friendsActor)));
            _userPresenceManager.Tell(new SubscribeToMyFriendsPresenceChanges(userName, frie
            _userPresenceManager.Tell(new FetchAllPresenceInformationForMyFriends(userName))
        }
    });

    // other Receive methods removed for brevity
}
```

The `ServerSessionManager` receives the `BootstrapFriendsPresence` message and will lazily create a `FriendsPresenceActor` - and it will immediately tell that `FriendsPresenceActor` to subscribe the `FriendsUIActor` (which exists over the network in a totally different application!) to any future presence updates.

The `ServerSessionManager` also tells the `FriendsPresenceActor` to send *all current friends presence information for this user*.

```
/// <summary>
/// Session-specific actor for authorized sessions.
///
/// Publishes updates and changes related to the availability of friends
/// back to individual connected clients.
/// </summary>
public class FriendsPresenceActor : ReceiveActor
{
    private readonly IActorRef _presenceActor;
    private readonly IActorRef _friendsListActor;
    private readonly Action<IActorRef, IActorRef, object> _publish;
    private readonly string _userName;
    private readonly HashSet<IActorRef> _subscribers = new HashSet<IActorRef>();

    /*
     * Used primarily for cleaning up subscriptions with the presence actor
     */
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
    private readonly HashSet<string> _friends = new HashSet<string>();

    public FriendsPresenceActor(string userName,
        Action<IActorRef, IActorRef, object> publish,
        IActorRef presenceActor, IActorRef friendsListActor)
    {
        _publish = publish;
        _presenceActor = presenceActor;
        _friendsListActor = friendsListActor;
        _userName = userName;

        Receive<SubscribeToMyFriendsPresenceChanges>(changes =>
        {
            _subscribers.Add(changes.Subscriber);
        });

        Receive<UnsubsribeFromMyFriendsPresenceChanges>(changes =>
        {
            _subscribers.Remove(changes.Subscriber);
        });

        Receive<Friend>(friend =>
        {
            _friends.Add(friend.FriendName);
            NotifySubscribers(friend);
            _presenceActor.Tell(new SubscribePresence(Self, friend.FriendName));
            _presenceActor.Tell(new PresenceActor.FetchPresence(friend.FriendName));
        });

        Receive<FriendRemoved>(removed =>
        {
            _friends.Remove(removed.FriendName);
            NotifySubscribers(removed);
            _presenceActor.Tell(new UnsubscribePresence(Self, removed.FriendName));
        });

        Receive<UserConnectivityInfo>(info => NotifySubscribers(info));

        Receive<FriendResponse>(response =>
        {
            NotifySubscribers(response.Friend);
        });

        Receive<FriendsResponse>(response =>
        {
            foreach(var friend in response.AllFriends)
                NotifySubscribers(friend);
        });

        Receive<FetchAllPresenceInformationForMyFriends>(friends =>
        {
            _presenceActor.Tell(new PresenceActor.FetchPresenceBatch(_friends.ToList()));
        });
    }

    protected override void PreStart()
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
    {
        _friendsListActor.Tell(new GetFriends(_userName));
        _friendsListActor.Tell(new SubscribeFriends(Self, _userName));
    }

    protected override void PostStop()
    {
        /*
         * Unsubscribe from all messages
         */
        foreach(var f in _friends)
            _presenceActor.Tell(new UnsubscribePresence(Self, f));

        _friendsListActor.Tell(new UnsubscribeFriends(Self, _userName));
    }

    private void NotifySubscribers(object message)
    {
        foreach (var sub in _subscribers)
            _publish(sub, Self, message);
    }
}
```

The `FriendsPresenceActor` will handle the `SubscribeToMyFriendsPresenceChanges` and subscribe the `IActorRef` belonging to the `FriendsUIActor` to its list of `_subscribers`. After doing that, the `FriendsPresenceActor` will subscribe to presence updates from the `PresenceActor`, a global actor that acts as the single source of truth for the presence of all users by listening for successful user authentications from each `ServerSessionManager`.

The `FriendsPresenceActor` will receive a `UserConnectivityInfo` message for each friend belonging the logged in user, and it will publish those `UserConnectivityInfo` messages to the `FriendUIActor` on the `AkkaTime.Client`.

```
/* Receive method inside AkkaTime.Client.FriendsUIActor */
Receive<UserConnectivityInfo>(connectivityInfo =>
{
    if (_friends.ContainsKey(connectivityInfo.Status.UserName))
    {
        var friend = _friends[connectivityInfo.Status.UserName];
        friend.SetConnectivity(connectivityInfo);
    }
    else
    {
        AddFriend(new Friend(connectivityInfo.Status.UserName), connectivityInfo);
    }
});
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Each time a `UserConnectivityInfo` message arrives, the `FriendsUIActor` will update its user interface and its peer list for knowing how to establish peer-to-peer connections between AkkaTime clients.

This techique allows up-to-date information to be propagated *fast as the network and application can possibly allow*. The `FriendsUIActor` is designed under the assumption that it always has the most recent data about the status of online, offline, or idle friends - so latency isn't even a concern for us because *that's information we don't have to fetch over the network. Instead, it's pushed into us.*

Timeouts

This section will be brief, because timeouts are trivially easy to implement in Akka.NET - but one thing we don't often do is really question their purpose.

> It's the job of a request timeout to set an upper bound on how much latency we're willing to tolerate for a particular request before it becomes economical for our applciation to fail the timed-out request in order to remain responsive for others.

So we can use timeouts for cross-network calls as a means of protecting the sender side of a networked application from deadlocking on a request that can never be completed.

```
/* Inside AkkaTime.Common.SecureActors.SessionHandshakeActor */
private void WaitForHandshake()
{
    _currentState = HandshakeState.Waiting;
    if (_isInbound)
    {
        // INBOUND BEHAVIORS
        Receive<HandshakeInfo>(info =>
        {
            _authenticator.Tell(new VerifySession(info.Session));
            _currentState = HandshakeState.WaitingNotVerified;
            _remoteSessionActor = info.SessionActor; //keep track of the remote session acto
            _remoteSessionHandshakeActor = info.SesssionHandshakeActor;
            _remoteSession = info.Session;
            SetReceiveTimeout(_sessionVerifyTimeout);
        });

        Receive<ReceiveTimeout>(timeout => DenyHandshakeAndShutDown("Timeout during identity

        Receive<ISessionVerificationResult>(result =>
        {
            if (result.Verified) //successful verification!
            {
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
                _currentState = HandshakeState.WaitingVerified;
                SetReceiveTimeout(null);
               _sessionManager.StartSecureSessionInbound(new InboundCreateSecureSession(_rem
                    _remoteSessionActor)).PipeTo(Self);

            }
            else //unsuccessful verification!
            {
                DenyHandshakeAndShutDown(string.Format("Unable to verify session {0} for use
                    _remoteSession.SessionId, _remoteSession.UserName));
            }
        });

        //other Receive methods omitted for brevity
    }
}
```

Inside AkkaTime's `SessionHandshakeActor`, the actor responsible for initiating remote communication over the network and exchanging AES keys in order to start encrypted communications, the "inbound" `SessionHandshakeActor` defines a maximum allowable timeout for verifying the identity of the `ActorSystem` on the outbound side of the association.

There are multiple ways to implement timeout behavior in Akka.NET - in this case we use the `SetReceiveTimeout` method. If a `ReceiveTimeout` message arrives before an `ISessionVerificationResult` does, then the `SessionHandshakeActor` times this operation out and cancels the handshake process.

We do this on the server side so we don't have tons of parallel handshakes that are just "hung" taking up threads, and we do this on the client-side so we can notify the end user that there might be connectivity issues with the server. The user has the option of retrying their request or quitting the `AkkaTime.Client`.

Detecting and Handling Network Failures

The next fun type of network problem we're going to work on is the infamous "unreliable network" problem. Networks sometimes drop messages and sometimes you might even get accidentally dropped from the network altogether! How can we use Akka.NET to be resilient to these types of failures?

In this section we introduce two patterns for dealing with this specific issue:

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

1. **AtLeastOnceDeliveryActor** - a special actor inside Akka.Persistence who queue messages to a durable store and sends them until receiving explicit acknowledgement from the intended recipient.
2. **Retry / Reconnect** - a pattern we can use to pro-actively reestablish communication with a remote process.

`AtLeastOnceDeliveryActor` Technique for Reliable Message Delivery

The last thing we want to have happen when we're trying to send important messages like e-commerce orders or financial data is have messages get lost in transit over the network. But sometimes the network fails! A switch could burn out at exactly the wrong moment; a technician in an important data center could accidentally unplug the router your packet was sitting on; and a million other possible ways your message could fail to get delivered.

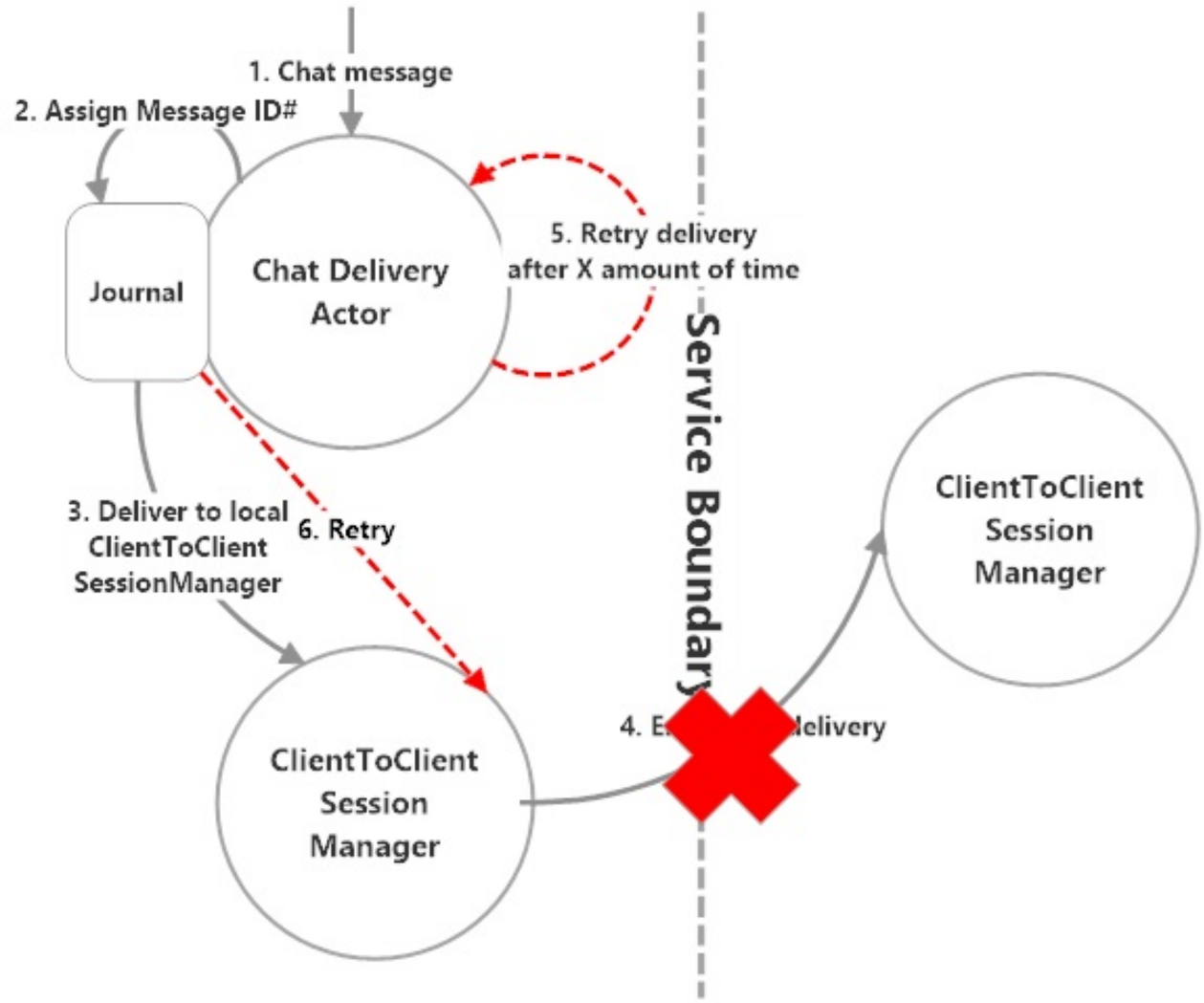
So what do we do? We can use the `Akka.Persistence.AtLeastOnceDeliveryActor` to do the following:

1. Log messages to a durable store, so they can be recovered in the event of a restart or process crash;
2. Deliver messages to an intended recipient, in this case an actor over the network; and
3. Re-deliver messages until receiving an acknowledgement by the recipient that each individual message has been received.

Imagine the following scenario within `AkkaTime.Client`: we try to deliver a chat message to another peer, but the message doesn't get delivered for some reason. Maybe the actor on the other end of the network dies or maybe the network gets interrupted temporarily .

We can use created the `ChatDeliveryActor`, a child class that implements all of the abstract methods from `AtLeastOnceDeliveryActor`, and it does the following during a failure scenario.

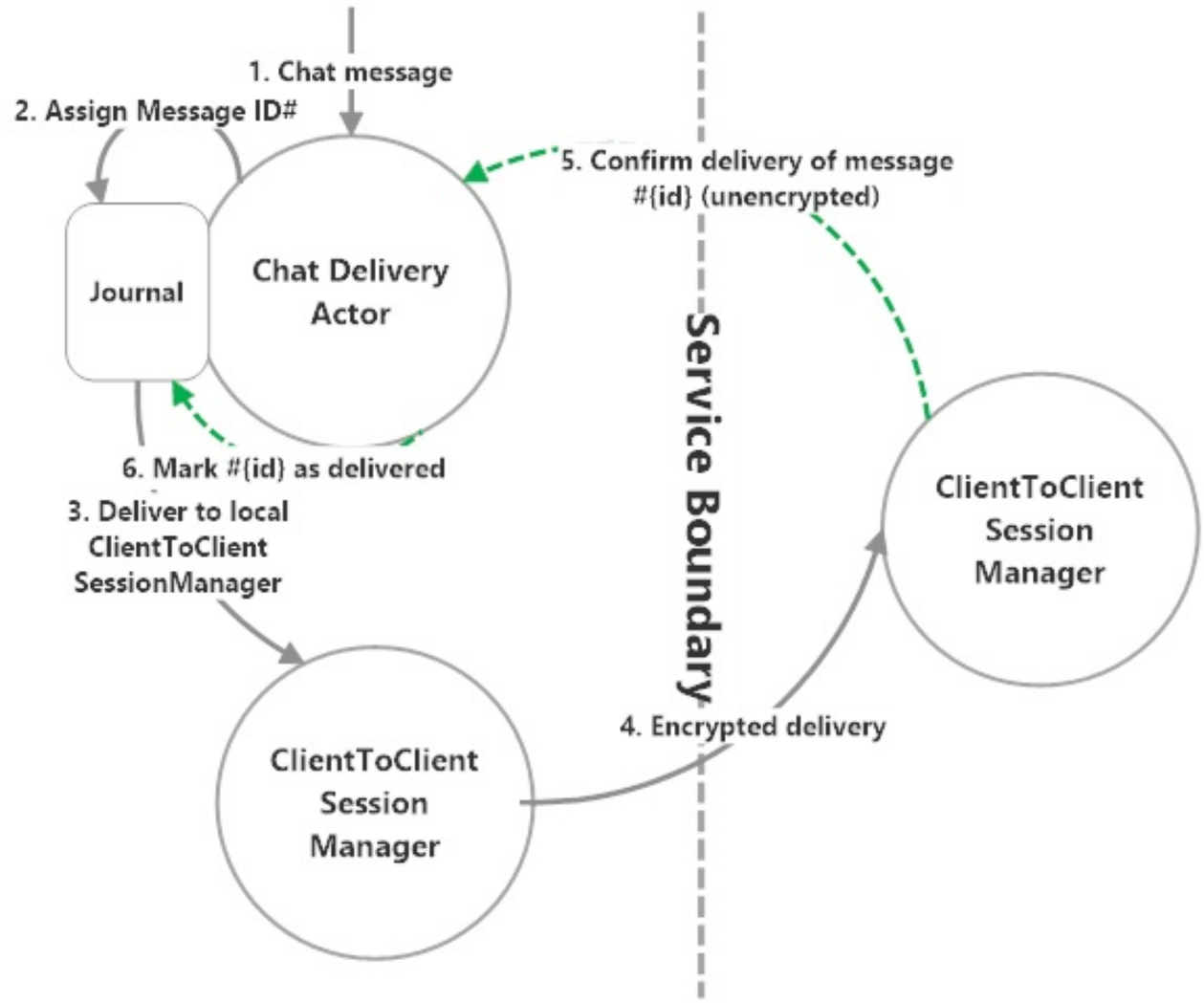
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



The `ChatDeliveryActor` will keep retrying to deliver its messages until it receives an acknowledgment. But one other thing that is important to point out is that the `ChatDeliveryActor` assigns a *id number* to each outgoing message. This is how the plumbing built into the `AtLeastOnceDeliveryActor` is able to verify that individual messages have been sent or received.

Here's what this interaction between actors looks like when the messages delivered by the `ChatDeliveryActor` go through successfully over the network.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



In this scenario, the remote `ClientToClientSessionManager` on the other side of the network successfully receives the message and sends back a user-defined "confirmation" message directly to the `ChatDeliveryActor`. And when the `ChatDeliveryActor` receives a confirmation message, it's able to mark the message as "delivered" and move on.

Let's see what this looks like in code!

```
/// <summary>
/// Used to guarantee delivery of messages to other chat clients
/// </summary>
public class ChatDeliveryActor : AtLeastOnceDeliveryActor
{
    private readonly IActorRef _clientEndpoint;
    private readonly IActorRef _chatUIActor;
    private readonly Guid _chatId;

    public ChatDeliveryActor(IActorRef clientEndpoint, IActorRef chatUIActor)
    {
        _clientEndpoint = clientEndpoint;
        _chatUIActor = chatUIActor;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        _chatId = Guid.NewGuid();
    }

    protected override bool ReceiveRecover(object message)
    {
        if (message is ChatMessage)
        {
            var chat = message as ChatMessage;
            Deliver(_clientEndpoint.Path, id =>
                new ClientToClientSessionManager.SendChatMessage(new ChatDelivery(id, chat))
        }
        else if (message is ChatConfirmation)
        {
            var deliveryId = ((ChatConfirmation) message).ChatId;
            ConfirmDelivery(deliveryId);
        }
        else
            return false;
        return true;
    }

    protected override bool ReceiveCommand(object message)
    {
        var chat = message as ChatMessage;
        if (chat != null)
        {
            Persist(chat, chatMessage =>
            {
                Deliver(_clientEndpoint.Path, id =>
                {
                    return new ClientToClientSessionManager.SendChatMessage(new ChatDelivery
                });
            });
        }
        else if (message is ChatConfirmation)
        {
            var deliveryId = ((ChatConfirmation)message).ChatId;
            ConfirmDelivery(deliveryId);
        }
        else
            return false;
        return true;
    }

    public override string PersistenceId
    {
        get { return _chatId.ToString(); }
    }
}
```

We don't really care about the `PersistenceId` for the `ChatDeliveryActor`, because we're only serving up messages from the in-memory journal for chat sessions that are fairly

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

short-lived. So that's why we use a random `Guid` for the `PersistenceId`.

That being said, we have a few message types that are important:

- `ChatMessage` - this is the raw data that is passed from the `ChatForm` to the `ChatDeliveryActor` and is used by the `ChatUIActor` to display messages onscreen.
- `ChatDelivery` - this is a wrapper for a `ChatMessage` plus the message ID number assigned by the `AtLeastOnceDeliveryActor` for tracking which messages still need to be confirmed by the sender.
- `ClientToClientSessionManager.SendChatMessage` - this is just a wrapper for the `ClientToClientSessionManager` so it can distinguish outbound messages from inbound messages. Inbound messages arrive as just `ChatDelivery` objects, whereas outbound messages are wrapped inside this class.
- `ChatConfirmation` - confirmation message that contains the ID number of the `ChatDelivery` that is being acknowledged by the receiver.

Here's what those classes look like:

```
/* Inside AkkaTime.Common.Messages.Chat.ChatMessage.cs */

/// <summary>
/// Used for guaranteed delivery of chat messages
/// </summary>
public class ChatDelivery
{
    public ChatDelivery(long chatId, ChatMessage message)
    {
        Message = message;
        ChatId = chatId;
    }

    public long ChatId { get; private set; }
    public ChatMessage Message { get; private set; }
}

public class ChatConfirmation
{
    public ChatConfirmation(long chatId)
    {
        ChatId = chatId;
    }

    public long ChatId { get; private set; }
}

/// <summary>
/// A chat message delivered over the network from one peer to another
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
/// </summary>
public class ChatMessage
{
    public ChatMessage(string sender, string message, DateTimeOffset when)
    {
        When = when;
        Message = message;
        Sender = sender;
    }

    [Required]
    public string Sender { get; private set; }

    [Required]
    public string Message { get; private set; }

    public DateTimeOffset When { get; private set; }
}

/* Inside AkkaTime.Client.ClientToClientSessionManager */
/// <summary>
/// Send an outbound message to the receiver
/// </summary>
public class SendChatMessage
{
    public SendChatMessage(ChatDelivery chat)
    {
        Chat = chat;
    }

    public ChatDelivery Chat { get; private set; }
}
```

So when we want to send a message, the `ChatWindow`'s will call the following code after the **Send** button is clicked.

```
/* In AkkaTime.Client.ChatWindow.cs */

private void TrySendMessage()
{
    if (tbChatMessage.Text != null)
    {
        _chatDeliveryActor.Tell(
            new ChatMessage(_localSession.UserName, tbChatMessage.Text,
                DateTimeOffset.Now));
        tbChatMessage.Text = "";
    }
}
```

We send a new `ChatMessage` to the `ChatDeliveryActor` directly from the form, and the

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

`ChatDeliveryActor` runs the following bit of code inside its `ReceiveCommand` method:

```
var chat = message as ChatMessage;
if (chat != null)
{
    Persist(chat, chatMessage =>
    {
        Deliver(_clientEndpoint.Path, id =>
        {
            return new ClientToClientSessionManager.SendChatMessage(new ChatDelivery(id, cha
        });
    });
}
```

We do a couple of things here:

1. Call `Persist` to save the original `ChatMessage` to this actor's journal, so it can be recovered later if the actor died before it had a chance to confirm delivery of the message.
2. Call `Deliver` to have the message delivered to the `ActorPath` of the `ClientToClientSessionManager` - the reason why we deliver to an `ActorPath` and not an `ActorSelection` is so this delivery can be re-done in the event of a total system restart, and since this `AtLeastOnceDeliveryActor`s could techncially be recreated with the same `PersistenceId` but in a different place in the actor hierarchy, an `ActorSelection` with relative paths would fail to resolve.

When the `Deliver` method gets called it generates a new `long` inside the lambda on the right. This `long` is the message ID and we pass it into the constructor of the `ChatDelivery` message along with the original `ChatMessage`. This content is then delivered to the `ClientToClientSessionManager`.

```
/* Inside AkkaTime.Client.ClientToClientSessionManager */
private void Chatting()
{
    Receive<ChatDelivery>(chat =>
    {
        Sender.Tell(new ChatConfirmation(chat.ChatId));
        _chatUIActor.Forward(chat.Message);
    });

    Receive<SendChatMessage>(send =>
    {
        Collection<ValidationResult> results;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        var isValid = AnnotationsValidator.TryValidate(send.Chat, out results);
        if (isValid)
        {
            _chatUIActor.Tell(send.Chat.Message);
            _securePublisher.Tell(new SecurePublish(RemoteSessionActor, Sender, send.Chat));
        }
        else
        {
            _chatUIActor.Tell(new FailedChatMessage(send.Chat.Message, results.ToErrorMessag
            Sender.Tell(new ChatConfirmation(send.Chat.ChatId)); //confirm delivery of the f
        }
    });

    // other Receive methods omitted for brevity
}
```

Inside these two `Receive<T>` methods inside the `ClientToClientSessionManager` we can see both the code for outbound message delivery to the remote system and the inbound message processing system.

First, for **outbound messages** (`Receive<SendChatMessage>`), we perform some validation in order to make sure that this is a valid chat message. In the event that it's an *invalid* chat message we send a `FailedChatMessage` back to the `ChatUIActor`, which will subsequently be printed on-screen. We will also send back a `ChatConfirmation` message to the local `ChatDeliveryActor` so it knows to stop trying to deliver the invalid `ChatMessage` over the network.

In the event of a successful validation effort, the local `ClientToClientSessionManager` will deliver just the `ChatDelivery` message to the remote `ClientToClientSessionManager` on the other end of the association.

For **inbound messages** (`Receive<ChatDelivery>`) the `ClientToClientSessionManager` will send the remote `ChatDeliveryActor` a `ChatConfirmation` and forward the `ChatMessage` onto the `ChatUIActor`.

And going back to the `ChatDeliveryActor`, here's where it processes the `ChatConfirmation` messages it receives from both local and remote senders inside its `ReceiveCommand` method.

```
else if (message is ChatConfirmation)
{
    var deliveryId = ((ChatConfirmation)message).ChatId;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        ConfirmDelivery(deliveryId);
    }
```

We call the `AtLeastOnceDeliveryActor`'s built-in `ConfirmDelivery` method and pass in the ID # of the messagewe're delivering. Once this occurs, the `ChatDeliveryActor` considers the delivery of this message to be complete and forgets about this message.

In the event that the `ConfirmDelivery` message isn't received within a reasonable period of time (the default period is 5 seconds), the `AtLeastOnceDeliveryActor` will automatically attempt to re-deliver the message.

You can modify the delivery interval by changing the following setting inside HOCON configuration.

```
# Interval between redelivery attempts
akka.persistence.at-least-once-delivery.redeliver-interval = 5s
```

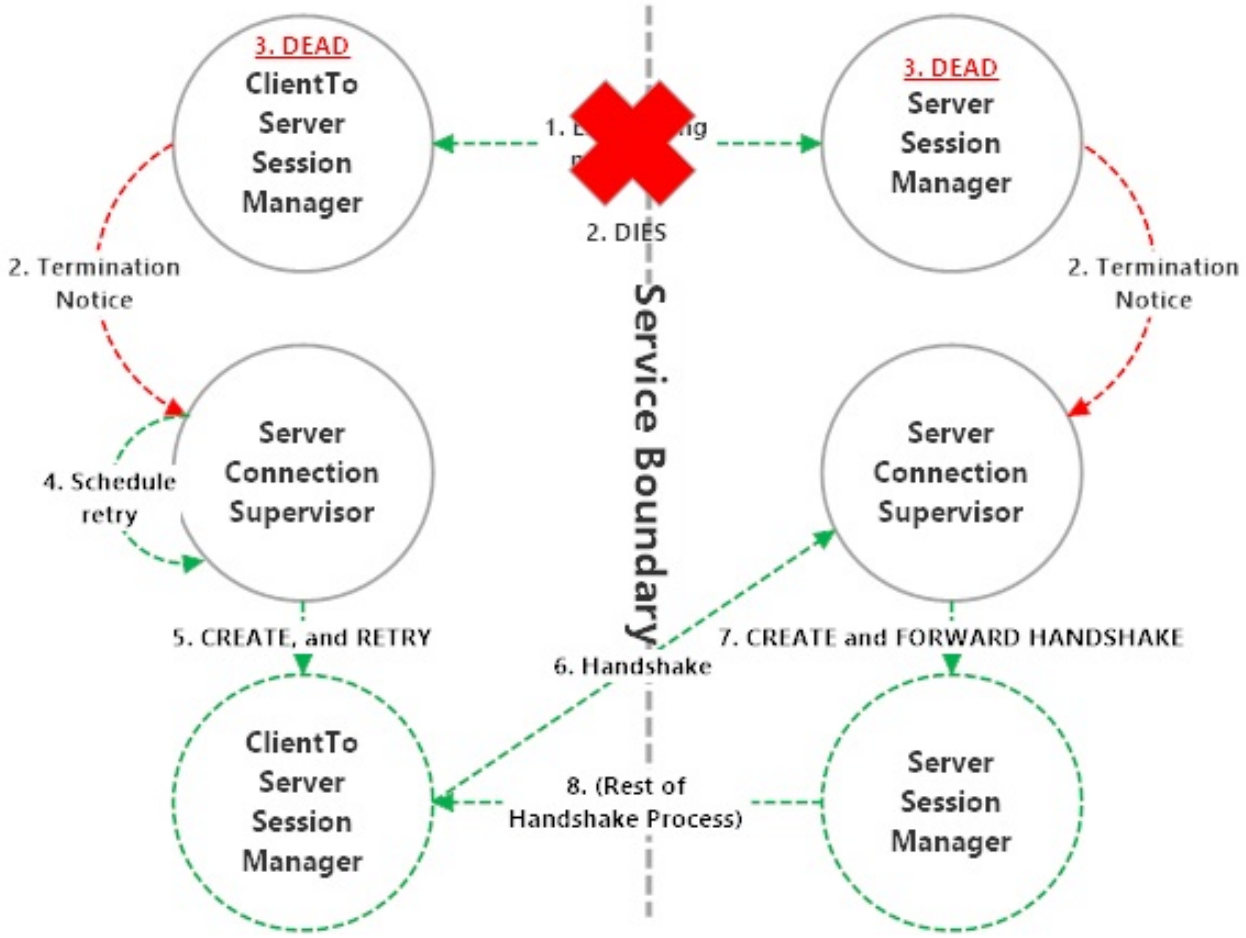
This will change the redeliver interval across the board for ALL `AtLeastOnceDeliveryActor`s.

Retry / Reconnect Technique

In the event of total network disruption, such as a socket connection accidentally being dropped due to a failure in network hardware, one of the things we can do is program some of our actors responsible for initiating outbound Akka.Remote communication with other remote systems to re-establish contact periodically in the event of a disconnection.

> **NOTE:** Akka.Cluster implements this technique to reconnect to nodes that have become unreachable, so if you're using that module then there's no need to do this yourself.

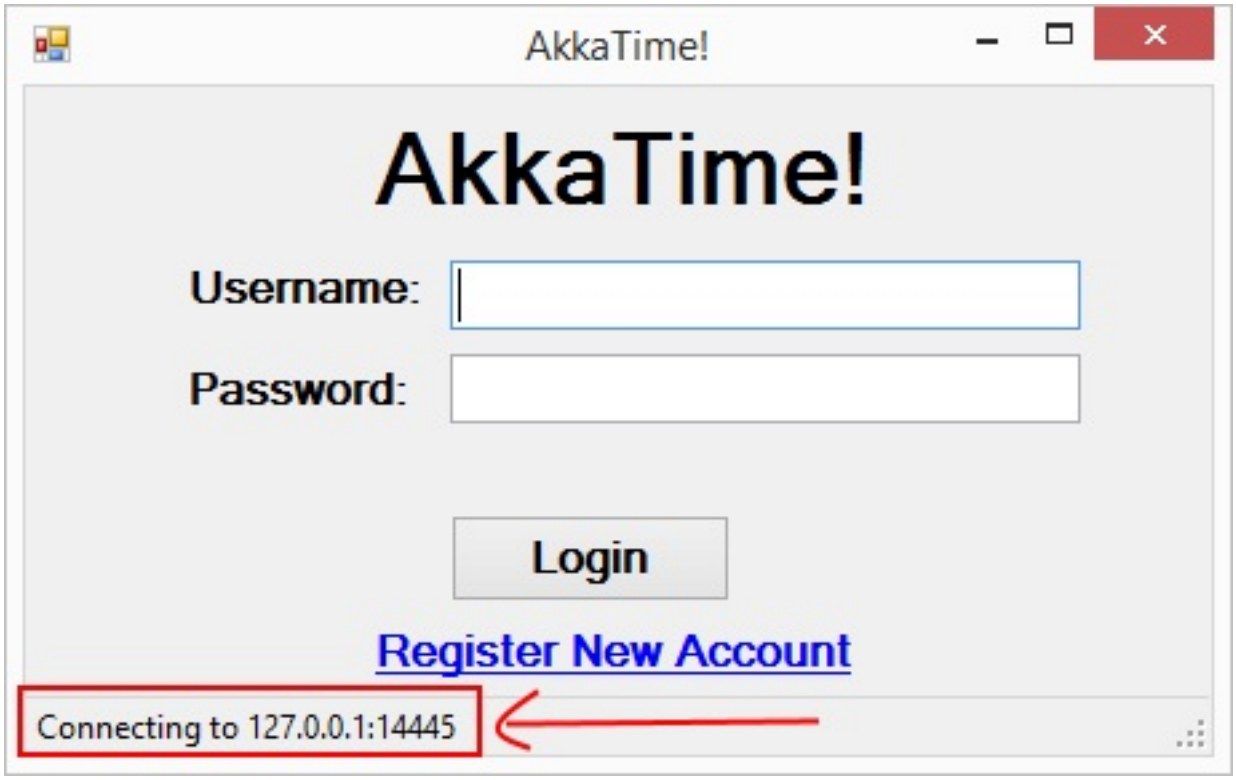
Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



In AkkaTime it's the responsibility of the client to establish connectivity with the server, so we implemented an actor who uses the retry / reconnect technique in the event that connectivity to the server is lost: the `ServerConnectionManager`.

If you run the AkkaTime sample you watch the `ServerConnectionManager` repeatedly attempt to contact the server on the client's Authentication and Friends screens.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



The process for implementing this technique is straightforward:

1. Create an actor responsible for listening for changes in connectivity (there's multiple ways of detecting disassociations, but my personal preference is to use the "death pact" pattern, which we'll reveal in a moment.)
2. Have this actor publish changes in connectivity to an actor who's responsible for initiating outbound remote connections - we'll call this our "connection supervisor."
3. Have the connection supervisor actor kick off the connection process and wait for hear for some response from the other end of the connection.
4. Rinse and repeat step 3 until we totally give up on the remote host or until we successfully connect.

Let's explore how we did this inside AkkaTime.

```
/* Inside AkkaTIme.Client.SessionConnectivityStatusActor.cs */
/// <summary>
/// Actor responsible for managing local notifications about changes
/// in connectivity to the server
/// </summary>
public class ServerConnectivityStatusActor : ReceiveActor
{
    private ServerConnectivityStatus _status;
    private Address _serverAddress;
    private readonly HashSet<IActorRef> _subscribers = new HashSet<IActorRef>();
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
    public ServerConnectivityStatusActor(Address serverAddress,
    ServerConnectivityStatus status = ServerConnectivityStatus.Disconnected)
    {
        _status = status;
        _serverAddress = serverAddress;

        Receive<SubscribeServerConnectivity>(connectivity =>
        {
            _subscribers.Add(connectivity.Subscriber);
        });

        Receive<UnsubscribeServerConnectivity>(connectivity =>
        {
            _subscribers.Remove(connectivity.Subscriber);
        });

        Receive<GetConnectivity>(connectivity =>
            Sender.Tell(new ConnectivityNotification(_serverAddress,_status)));

        Receive<ConnectTo>(connect =>
        {
            _status = ServerConnectivityStatus.Connecting;
            PublishToSubscribers();
        });

        Receive<SessionOpened>(opened =>
        {
            _status = ServerConnectivityStatus.Connected;
            PublishToSubscribers();
        });

        Receive<SessionFailedToOpen>(open =>
        {
            _status = ServerConnectivityStatus.UnableToConnect;
            PublishToSubscribers();
        });

        Receive<SessionClosed>(sessionClosed =>
        {
            _status = ServerConnectivityStatus.Disconnected;
            PublishToSubscribers();
        });
    }

    private void PublishToSubscribers()
    {
        var message = new ConnectivityNotification(_serverAddress,_status);
        foreach(var sub in _subscribers)
            sub.Tell(message);
    }
}
```

The first actor we created is the `ServerConnectivityStatusActor`, an actor whose only

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

responsibility is to listen to and publish changes in server connectivity.

```
/* Inside AkkaTIme.Client.ServerConnectionSupervisor.cs */
// <summary>
/// Actor responsible for restarting connections to the server endpoint in the event of a di
/// </summary>
public class ServerConnectionSupervisor : ReceiveActor
{
    #region Internal message classes

    /// <summary>
    /// Used to bootstrap a connection to the server
    /// </summary>
    public class ConnectToServer { }

    #endregion

    //The credentials we can use to restart our connection to the server
    private AuthenticationChallenge _successfulAuthenticationChallenge;
    private IActorRef _serverConnectivityStatus;
    private ICancelable _connectRetry;
    private readonly Address _serverAddress;
    private readonly byte[] _cryptKey;

    private IActorRef _sessionManager;

    public ServerConnectionSupervisor(IActorRef serverConnectivityStatus, Address serverAddr
    {
        _serverConnectivityStatus = serverConnectivityStatus;
        _serverAddress = serverAddress;
        _cryptKey = cryptKey;

        _serverConnectivityStatus.Tell(new SubscribeServerConnectivity(Self));
        Self.Tell(new ConnectToServer());
        Disconnected();
    }

    private void Disconnected()
    {
        Receive<ConnectToServer>(server =>
        {
            var session = ActiveSession.AnonymousSession;
            _sessionManager =
                ActorSystemRefs.ClientActor =
                    Context.ActorOf(
                        Props.Create(
                            () => new ClientToServerSessionManager(_cryptKey, session)),
                            ActorPaths.ClientEndpoint.Name);
            var sessionManagerWrapper = new SessionManagerWrapper(false, _sessionManager);
            var handshakeActor =
                Context.ActorOf(SessionHandshakeActor.OutboundProps(sessionManagerWrapper, s
                    (Context.System.Settings.Config.GetTimeSpan("akkatime.sessions.handshake
                        TimeSpan.FromSeconds(4.0)))));
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            //subscribe the connectivity actor to all lifecycle events, including debug even
            _sessionManager.Tell(new SubscribeToSessionLifecycleEvents(_serverConnectivitySt

            var remoteEndpoint =
                Context.ActorSelection(new RootActorPath(_serverAddress) / "user" / ActorPat
            var connectTo = new ConnectTo(remoteEndpoint, BeginHandshake.Instance);

            _serverConnectivityStatus.Tell(connectTo);
            handshakeActor.Tell(connectTo);
            BecomeConnecting();
        });
    }

    private void BecomeConnecting()
    {
        if (_connectRetry != null)
        {
            _connectRetry.Cancel();
            _connectRetry = null;
        }
        Become(Connecting);
    }

    private void Connecting()
    {
        Receive<AuthenticationChallenge>(challenge =>
        {
            _successfulAuthenticationChallenge = challenge;
        });

        Receive<ConnectivityNotification>(status =>
        {
            if (status.ConnectivityStatus == ServerConnectivityStatus.Disconnected
                || status.ConnectivityStatus == ServerConnectivityStatus.UnableToConnect)
            {
                BecomeDisconnected();
            }
            else if(status.ConnectivityStatus == ServerConnectivityStatus.Connected)
            {
                BecomeConnected();
            }
        });
    }

    private void BecomeDisconnected()
    {
        Become(Disconnected);
        Context.Stop(_sessionManager);
        _connectRetry = Context.System.Scheduler
            .ScheduleTellRepeatedlyCancelable(TimeSpan.FromSeconds(3.0),
            TimeSpan.FromSeconds(10.0), Self, new ConnectToServer(), ActorRefs.NoSender);
    }

    private void BecomeConnected()
    {
        if (_connectRetry != null)
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        {
            _connectRetry.Cancel();
            _connectRetry = null;
        }
        Become(Connected);
        Context.ActorSelection(ActorPaths.AuthHandler.Path).Ask<ActorIdentity>(new Identify(
            tr =>
            {
                var actor = tr.Result.Subject;
                _sessionManager.Tell(new SubscribeToSessionLifecycleEvents(actor, false), ac
                _sessionManager.Tell(new GetCurrentLifecycleStage(), actor);
                if (_successfulAuthenticationChallenge != null)
                {
                    actor.Tell(_successfulAuthenticationChallenge, ActorRefs.NoSender);
                }
            }, TaskContinuationOptions.ExecuteSynchronously);
    }

    private void Connected()
    {
        Receive<ConnectivityNotification>(status =>
        {
            if (status.ConnectivityStatus == ServerConnectivityStatus.Disconnected)
            {
                BecomeDisconnected();
            }
        });

        Receive<AuthenticationChallenge>(challenge =>
        {
            _successfulAuthenticationChallenge = challenge;
        });
    }
}
```

This is the entire source file for the `ServerConnectionSupervisor` - it does the following:

1. Begins in a `Disconnected` state and sends itself a `ConnectToServer` message, which in turn causes the `ServerConnectionSupervisor` to create a `ClientToServerSessionManager` and a `SessionHandshakeActor`. It's through these two actors that the attempt to connect to an `AkkaTime.Server` instance is done. Note that the `ServerConnectionSupervisor` subscribes itself to updates from the `ServerConnectivityStatusActor` and it has the newly created `ClientToServerSessionManager` publish its lifecycle messages to the `ServerConnectivityStatusActor`. These subscriptions are how the `AkkaTime.Client` is able to detect and react to network failures between itself and the server.
2. Once the `SessionHandshakeActor` is created and begins its communication process with the server, the `ServerConnectionSupervisor` transitions to a `Connecting` state and

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

waits for notifications about the status of its connection with the server.
   i. If the **connection to the server is successful**, it will transition to a `Connected` state and wait for further updates.
   ii. If the **connnection to the server is unsuccessful**, it will transition to a `Disconnected` state and schedule a retry attempt for 3 seconds in the future and then on a repeating loop once every 10 seconds.
3. If at any point the server gets disconnected from this particular client, the `ServerConnectionSupervisor` will transition to its `Disconnected` state and kick off a scheduled message to attempt to reconnect in 3 seconds.

These actors are both extremely simple, but effective in their abilities to reestablish downed connections if the server is able to begin receiving messages from the client again.

Detecting and Handling Failed Remote Processes

As if it wasn't bad enough that the network could fail on us, we also have to account for remote processes and computers failing too!

One of the advantages of Akka.NET is that it makes virtually no distinctions between a process failure and a network failure - **Akka.Remote treats a network failure as though the process itself was dead and automatically kills any remote-deployed actors and fires deathwatch notifications for any remote-watched actors on both ends of an association**.

As it turns out, we can use this to our advantage in the design of our applications!

Here are the three strategies we can use for making our applications resilient to failed processes:

1. **Death Pact Pattern** - actors on both ends of an association form an exclusive pair over the lifespan of their work and if one terminates, so does the other.
2. **Self-Cleaning Hierarchy Pattern** - used in combination with the death pact pattern, have entire actor hierarchies that build over the lifespan of a remote connection automatically clean themselves up in the event of a termination.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

Death Pact Pattern

The Death Pact Pattern is useful for when you have two actors who are "paired" for the lifecycle of some work that occurs over the network.

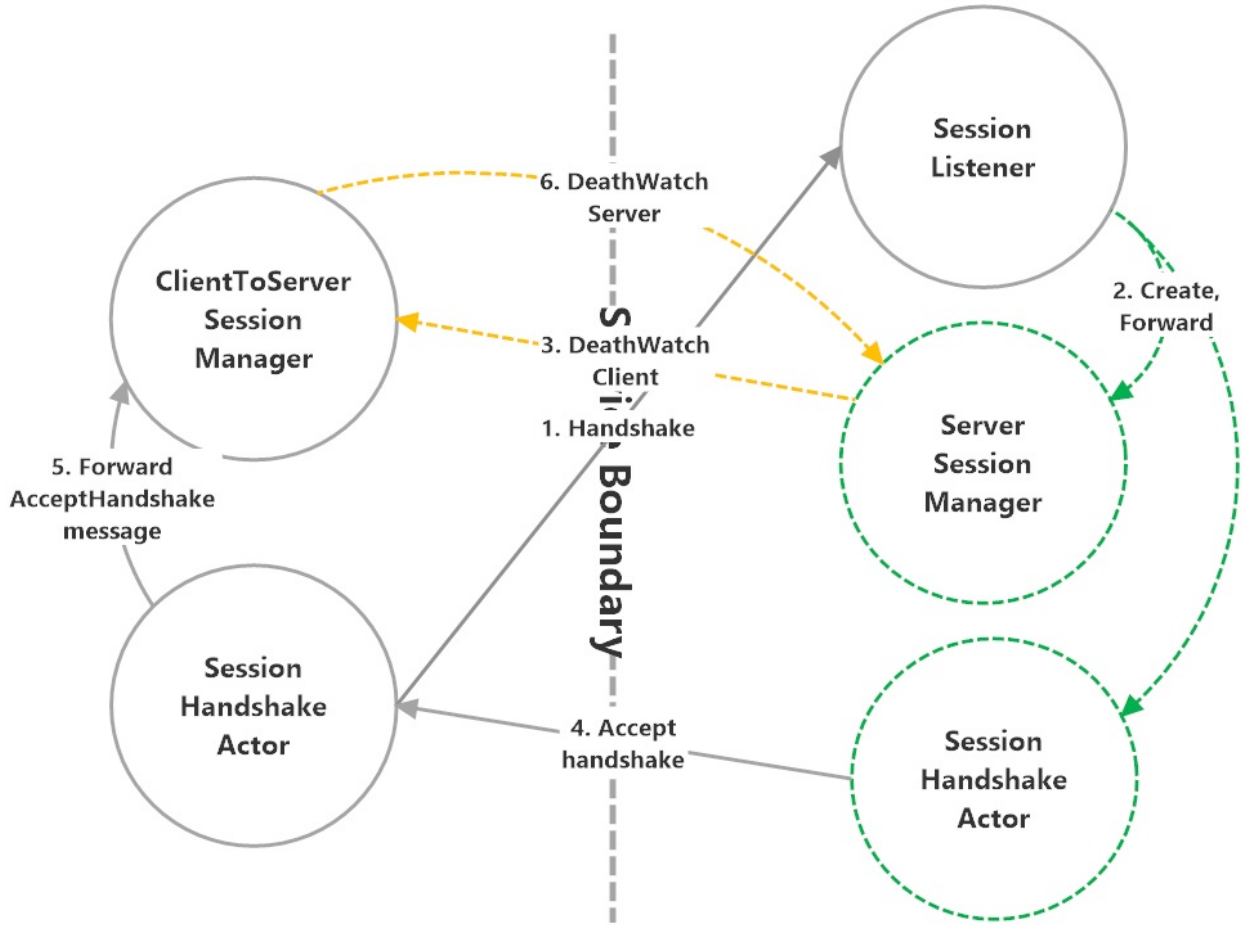
`DeathWatch` is a concept built into Akka.NET that allows actors to monitor the lifecycles of other actors who aren't necessarily their direct children. This tool is really helpful in cases where one actor's behavior depends on another specific actor instance, and it's especially helpful when the network sits between those actors.

Akka.Remote extends Akka.NET's `DeathWatch` capability to work seamlessly over the network, and `DeathWatch` notifications will fire for remote actors in each of the following circumstances:

1. The remote actor dies for any of the usual reasons, such as planned termination or the actor's parent being shut down, or
2. The association between the two actor systems is severed as a result of network or process issues.

Given this, we can use `DeathWatch` to help us know when and how to respond in the event of a remote Akka.NET process we depend on terminating - and we do this using the Death Pact pattern.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC



In AkkaTime, all actors who implement the `SessionManagerBase` class automatically subscribe to `DeathWatch` notifications for their counterpart on the opposite end of the association.

The `AkkaTime.Client` and the `AkkaTime.Server` instance both call `Context.Watch` on each other. That way, in the event that either the `ClientToServerSessionManager` or the `ServerSessionManager` dies OR the association between the two processes fail, both ends of the association can perform the necessary clean-up and respond.

Let's take a look at the portions of code within the `SessionManagerBase` class below that implement the Death pact pattern.

```
/* AkkaTime.Common.SecureActors.SessionManagerBase */

/// <summary>
/// Abstract base class for fulfilling message contracts expected by <see cref="SessionHands
///
/// The job of all <see cref="SessionManagerBase"/> implementations is to be able to manage
/// between two connected <see cref="ActorSystem"/> instances,
/// regardless of whether or not it's peer-to-peer or client-server.
/// </summary>
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
public abstract class SessionManagerBase : ReceiveActor
{
    protected readonly bool IsInbound;
    private bool _sessionOpen;

    protected IActorRef LocalSecureActor;
    protected IActorRef RemoteSecureActor;
    protected IActorRef RemoteSessionActor;

    protected readonly HashSet<IActorRef> Subscribers = new HashSet<IActorRef>();
    protected readonly HashSet<IActorRef> DebugSubscribers = new HashSet<IActorRef>();

    private object _currentSessionState;

    /// <summary>
    /// A reference to ourselves which can be accessed outside of the current <see cref="IAc
    /// </summary>
    protected readonly IActorRef SafeSelf = Context.Self;

    protected SessionManagerBase(bool isInbound)
    {
        IsInbound = isInbound;
        ReceiveHandshakes();
    }

    #region Receives

    /// <summary>
    /// Behavior during the handshake, for both the inbound and outbound side
    /// </summary>
    public void ReceiveHandshakes()
    {
        Receive<SessionHandshakeActor.InboundCreateSecureSession>(session =>
        {
            AssertInbound();
            PublishDebugEventToSubscribers(session);
            var sessionKey = GenerateSessionSecret();
            LocalSecureActor = CreateSecureActor(sessionKey);
            SetRemoteSessionActor(session.RemoteSessionManager);
            HandleRemoteSessionInformation(session.RemoteSession);
            Sender.Tell(new SessionHandshakeActor.InboundSecureSessionInformation(
                sessionKey, LocalSecureActor, GetLocalSessionInformation()));
        });

        Receive<SessionHandshakeActor.OutboundCreateSecureSession>(session =>
        {
            AssertOutbound();
            PublishDebugEventToSubscribers(session);
            RemoteSecureActor = session.RemoteSecureActor;
            SetRemoteSessionActor(session.RemoteSessionManager);
            HandleRemoteSessionInformation(session.RemoteSession);
            LocalSecureActor = CreateSecureActor(session.SharedSecret);
            Sender.Tell(new SessionHandshakeActor.OutboundSecureSessionInformation(LocalSecu
        });

        // Removed all receives not relevant to Death Pact pattern
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        LifeCycleSubscriptionReceives();
    }

    /// <summary>
    /// Handle all of the pub-sub stuff for lifecycle events here.
    /// </summary>
    protected void LifeCycleSubscriptionReceives()
    {
       // Removed all receives not relevant to Death Pact pattern

        /*
         * Might mean that the remote side of the connection died and we have to shut down.
         */
        Receive<Terminated>(notification =>
        {
            if (DidRemoteSessionActorDie(notification))
            {
                if (_sessionOpen)
                    DisconnectAndShutdown();
                else
                {
                    PublishToSubscribers(new SessionFailedToOpen("Remote endpoint terminated
                    Context.Stop(Self);
                }
            }
        });
    }

    private void DisconnectAndShutdown()
    {
        var remoteInfo = GetRemoteSessionInformation();
        PublishToSubscribers(new SessionClosed(GetLocalSessionInformation(), remoteInfo,
            string.Format("{0} disconnected.", remoteInfo.UserName)));
        Context.Stop(Self);
    }

    protected override void PostStop()
    {
        if (_sessionOpen)
        {
            var remoteInfo = GetRemoteSessionInformation();
            PublishToSubscribers(new SessionClosed(GetLocalSessionInformation(), GetRemoteSe
                string.Format("Ended session with {0}.", remoteInfo.UserName)));
        }
    }

    #endregion

    /// <summary>
    /// Set the remote session actor and subscribe for death watch notifications.
    /// </summary>
    /// <param name="remoteSessionActor"></param>
    protected void SetRemoteSessionActor(IActorRef remoteSessionActor)
    {
        RemoteSessionActor = remoteSessionActor;
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
        Context.Watch(RemoteSessionActor);
    }

    /// <summary>
    /// Determine whether or not the remote session actor died
    /// </summary>
    /// <param name="terminated">A deathwatch notifciation</param>
    /// <returns>true if the remote session actor died, false otherwise.</returns>
    protected bool DidRemoteSessionActorDie(Terminated terminated)
    {
        if (RemoteSessionActor == null) return false;
        return terminated.ActorRef.Equals(RemoteSessionActor);
    }
}
```

You can see that during the `Receive<SessionHandshakeActor.InboundCreateSecureSession>` and `Receive<SessionHandshakeActor.OutboundCreateSecureSession>` receives, which represent steps 3 and 6 of the above diagram respectively, that both receive methods each call `SetRemoteSessionActor`.

```
/// <summary>
/// Set the remote session actor and subscribe for death watch notifications.
/// </summary>
/// <param name="remoteSessionActor"></param>
protected void SetRemoteSessionActor(IActorRef remoteSessionActor)
{
    RemoteSessionActor = remoteSessionActor;
    Context.Watch(RemoteSessionActor);
}
```

`SetRemoteSessionActor` calls `Context.Watch`, which establishes the `Deathwatch` subscription on both ends of the network connection.

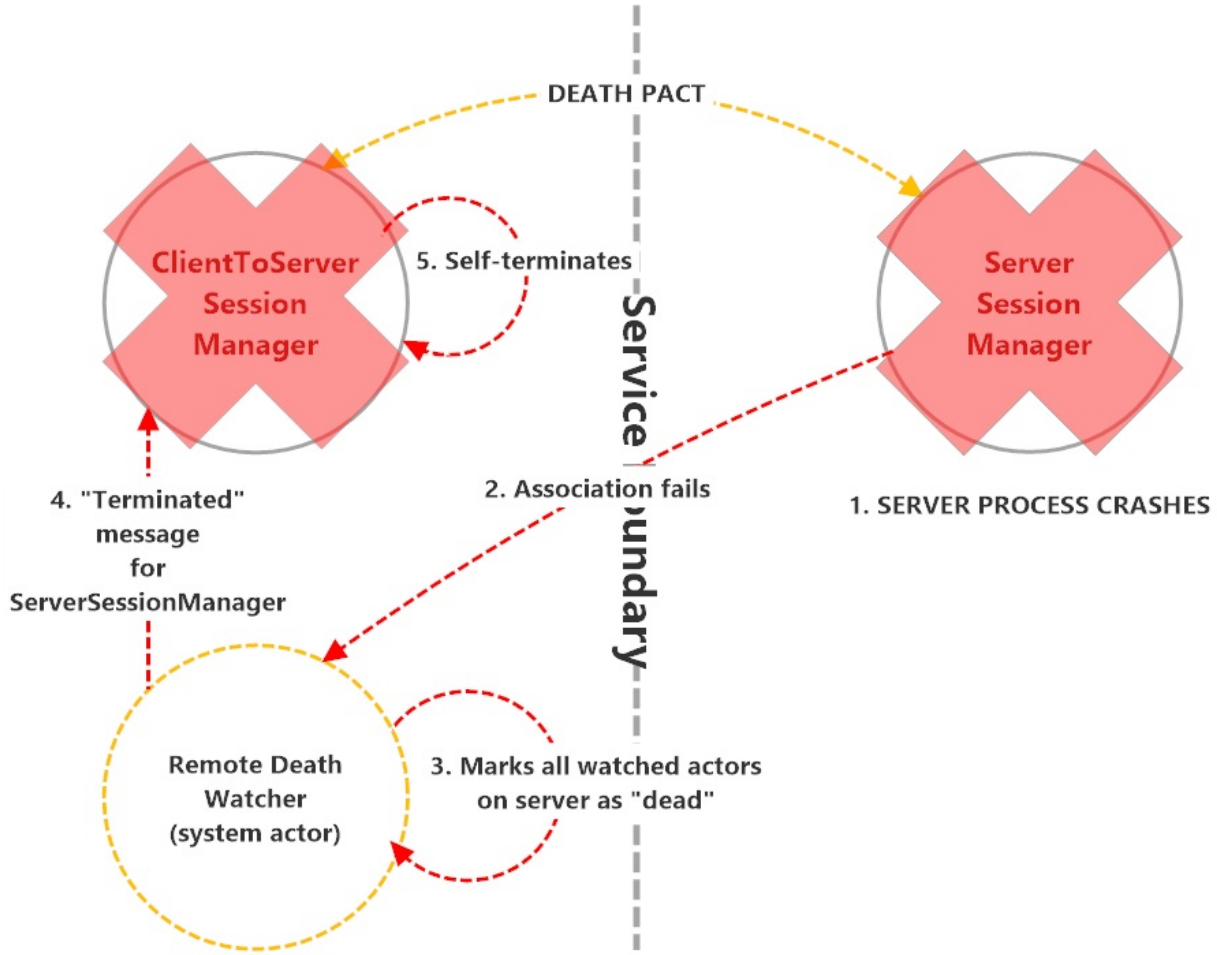
But in order for a `DeathWatch` subscription to work, we need to handle `Terminated` messages that are sent in the event that a `DeathWatch` event fires - so we do that inside the `LifeCycleSubscriptionReceives()` method.

```
Receive<Terminated>(notification =>
{
    if (DidRemoteSessionActorDie(notification))
    {
        if (_sessionOpen)
            DisconnectAndShutdown();
        else
```


Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

```
            {
                PublishToSubscribers(new SessionFailedToOpen("Remote endpoint terminated during
                Context.Stop(Self);
            }
        }
    });
```

When a `Terminated` message arrives, the first thing we need to do is see if this message was sent to this actor by the `RemoteSessionActor` on the other end of the association. If it's not from that actor, we can ignore it. But if it is, then we ourselves must shut down - hence the "death pact."



So when the `AkkaTime.Server` process crashes in this example, a `Terminated` message is generated for actor on `AkkaTime.Client` who was watching an actor on the server. When the `Terminated` message is delviered to the `ClientToServerSessionManager` it will check to verify that the `Terminated` message is for the `ServerSessionManager`.

If that's the case, then the `ClientToServerSessionManager` will shut itself down and notify all of its subscribers that the session with the server has been closed. This is the

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

mechanism we used to implement the Retry / Restart Pattern earlier - the `ServerConnectionSupervisor` gets notified that a disconnect has occurred and attempts to re-esablish an association with the configured `AkkaTime.Server` instance.

Different Types of `DeathWatch` Notifications

One important piece of data that the `Terminated` message gives is you some explanation as to WHY the actor you were watching has died.

```
public sealed class Terminated : IAutoReceivedMessage, IPossiblyHarmful
{
    public Terminated(IActorRef actorRef, bool existenceConfirmed, bool addressTerminated)
    {
        ActorRef = actorRef;
        ExistenceConfirmed = existenceConfirmed;
        AddressTerminated = addressTerminated;
    }

    public IActorRef ActorRef { get; private set; }


    public bool AddressTerminated { get; private set; }

    public bool ExistenceConfirmed { get; private set; }

    public override string ToString()
    {
        return "<Terminated>: " + ActorRef + " - ExistenceConfirmed=" + ExistenceConfirmed;
    }
}
```

The `Terminated.AddressTerminated` property returns **true** if the reason why we're receiving this death watch notification is because the association between our current `ActorSystem` and the remote `ActorSystem` that `Terminated.ActorRef` lived on failed.

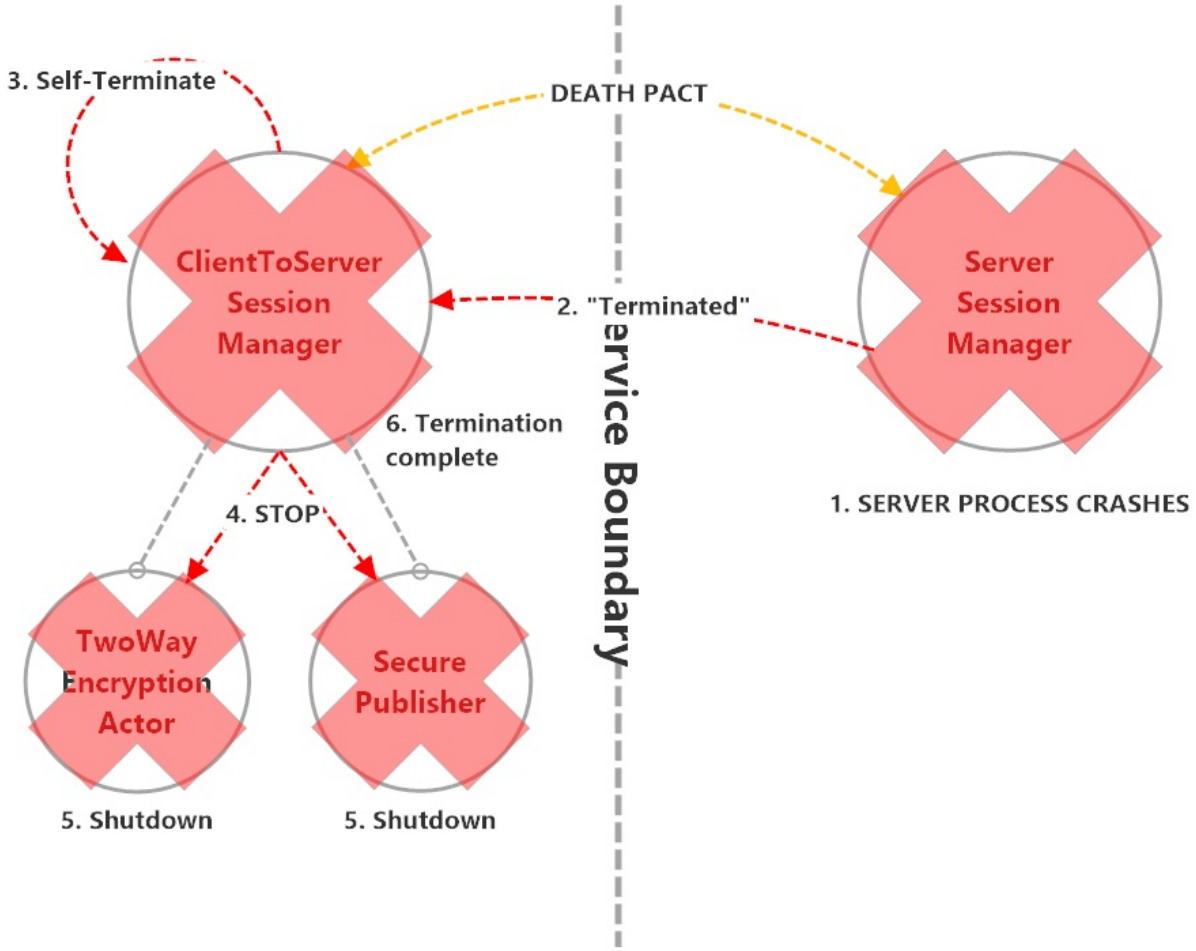
Therefore, when you implement the Death Pact Pattern you can use this data to distinguish between remote actors shutting down as a result of the network or as a result of application-specific termination.

Self-Cleaning Hierarchy Pattern

This pattern is almost used exclusively in combination with the Death Pact pattern. The Self-Cleaning Heirarchy Pattern is used to help clean up actors and

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

resources your application allocated in order to support remote actors on the other side of an association. In the event that the remote actors you were supporting finish their work and shut down cleanly; if they terminate unexpectedly; or if the network fails, you can use this pattern to clean up any resources on remote processes that were related to the watched actors.



There isn't any user-defined code required on our part to do this once we've implemented the Death Pact pattern. Both the `TwoWayEncryptionActor` and the `SecurePublisher` were created as a children of the `ClientToServerSessionManager`, thus they and any grand children they might have deployed will be killed as part of the `ClientToServerSessionManager`'s shut down logic.

If additional cleanup is required, such as removing subscriptons to other pub-sub actors who will still be alive once this cleanup is finished, you can override the `PostStop` method of each of these actors and include any relevant cleanup logic there.

Conclusion

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC

This marks the end of Petabridge's *Akka.NET Remoting* course! We hope you enjoyed it!

You should consider taking our other training courses:

- **Akka.NET Design Patterns** - learn how to model message-oriented systems and use battle-tested Akka.NET design patterns to build effective actor hierarchies.
- **Akka.NET Clustering** - learn how to build resilient self-healing Akka.NET clusters using Akka.Cluster. Perfect for developers interested in building distributed systems with Akka.NET.

Get in touch! => hi@petabridge.com
© 2015 Petabridge, LLC