Part II: Implementation of Networking

By Natallia Dzenisenka

Table Of Contents

Message serialization

To send any messages over the network, we need to have a serialization mechanism. Because F# works with .NET platform, it can utilize existing serialization libraries, such as Newtonsoft JSON.

In the NetworkServer namespace, there is a MessageSerialization module. The main functions we use are serializeNewtonsoft and deserializeNewtonsoft. The serialization function accepts a message object, converts it into JSON, and later converts it into a byte array. The deserialization function accepts the array of bytes, coverts it into a string and then to an object.

```
namespace NetworkServer

open Newtonsoft.Json
open System.Text

module MessageSerialization =

    let serializeNewtonsoft message = async {
        let settings = JsonSerializerSettings(TypeNameHandling =
TypeNameHandling.All, CheckAdditionalContent = true)
        return Encoding.ASCII.GetBytes(JsonConvert.SerializeObject(message,
settings))
    }

    let deserializeNewtonsoft fromBytes = async {
        let settings = JsonSerializerSettings(TypeNameHandling =
TypeNameHandling.All, CheckAdditionalContent = true)
        try
            return JsonConvert.DeserializeObject(Encoding.ASCII.GetString(fromBytes,
0, fromBytes.Length), settings)
        with
        | ex ->
            return failwith (sprintf "Exception deserializing object: %s."
ex.Message)
    }
```

There are other serialization libraries out there. Another great example is FsPickler, an easy approach for binary serialization. For example:

```
open MBrace.FsPickler

let serializePickle message = async {
    let binarySerializer = FsPickler.CreateBinarySerializer()
    return binarySerializer.Pickle message
}

let deserializePickle<'a> fromBytes = async {
    let binarySerializer = FsPickler.CreateBinarySerializer()
```

```
    return binarySerializer.UnPickle<'a> fromBytes
}
```

Network Server

What network protocol should we choose for a distributed system in F#?

Various distributed algorithms can require different distributed system models. In this project, we will primarily use UDP as a main underlying network protocol. However, some other algorithms might benefit from a TCP server, the implementation of which is provided for convenience.

In the namespace NetworkServer there's a module called Communication for hosting network communication abstractions. We'd like to be able to choose different implementations of underlying network communication. First of all, we define NetworkServer type, listing three main functions of a network server: StartServer, SendMessage, and ReceiveMessages. There can be more functions associated with a network server, but for the purposes of this example we rely on those defined.

```
namespace NetworkServer

module Communication =
    open System
    open System.Net
    open System.Net.Sockets

    [<AbstractClass>]
    type NetworkServer () =
        abstract member StartServer: Async<unit>
        abstract member SendMessage: obj -> string -> int -> Async<unit>
        abstract member ReceiveMessages: Async<unit>

    // ...
```

TCP server

Later in the module we define TcpServer type - implementation of a TCP server. The server can be started on a specified port. The server is also initialized with a given processMessage function. This function will be called after the server received a message.

```
namespace NetworkServer

module Communication =

    // ...

    type TcpServer(port:int, processMessage) =
        inherit NetworkServer()

        override x.SendMessage (message: obj) (toHost: string) (toPort: int) =
            async {
                let! messageBytes =
                    MessageSerialization.serializeNewtonsoft message
```

```
            use client = new TcpClient()
            client.Connect(IPAddress.Parse(toHost), toPort)
            use stream = client.GetStream()
            let size = messageBytes.Length
            let sizeBytes = BitConverter.GetBytes size
            do! stream.AsyncWrite(sizeBytes, 0, sizeBytes.Length)
            do! stream.AsyncWrite(messageBytes, 0, messageBytes.Length)
        }

    override x.ReceiveMessages = async {
        printfn "Listening for incoming TCP messages..."

        let listener = TcpListener(IPAddress.Loopback, port)
        listener.Start()

        while true do
            let client = listener.AcceptTcpClient()
            try
                let stream = client.GetStream()
                let sizeBytes = Array.create 4 0uy
                let! readSize = stream.AsyncRead(sizeBytes, 0, 4)
                let size = BitConverter.ToInt32(sizeBytes, 0)
                let messageBytes = Array.create size 0uy
                let! bytesReceived = stream.AsyncRead(messageBytes, 0, size)
                if bytesReceived <> 0 then
                    // Process message bytes using custom logic
                    do! processMessage messageBytes
            with
            | ex ->
                printfn "Exception receiving a TCP message: %s." ex.Message
    }

    override x.StartServer = async {
        printfn "Started a server on port %A." port
        do! x.ReceiveMessages
    }
```

UDP server

In the same module, we define UdpServer type - implementation of a UDP server. Similarly to the TcpServer type, it can be started on a specified port and pass received messages to the given processMessage function for further processing.

```
type UdpServer (port:int, processMessage) =
    inherit NetworkServer()

    override x.SendMessage (message: obj) (toHost: string) (toPort: int) =
        async {
            try
```

```
                let! messageBytes =
                    MessageSerialization.serializeNewtonsoft message

                let udpClient = new UdpClient()
                udpClient.Connect(toHost, toPort)

                udpClient.Send(messageBytes, messageBytes.Length) |> ignore
                udpClient.Close()
            with
            | ex ->
                printfn "Exception sending a UDP message: %s." ex.Message
        }

    override x.ReceiveMessages = async {
        printfn "Listening for incoming UDP messages..."
        let udpClient = new UdpClient(port)

        let receive =
            async {
                try
                    let remoteNode = IPEndPoint(IPAddress.Any, 0)
                    let messageBytes = udpClient.Receive(ref remoteNode)

                    // Process message bytes using custom logic
                    do! processMessage messageBytes
                with
                | ex ->
                    printfn "Exception receiving a UDP message: %s." ex.Message
            }

        while true do
            do! receive
    }

    override x.StartServer = async {
        printfn "Started a server on port %A." port
        do! x.ReceiveMessages
    }
```

Next

Take a look at the next section on Implementation of Failure Detectors.