Part III: Implementation of Failure Detectors

By Natallia Dzenisenka

Table Of Contents

Implementing Failure Detectors

In this part, we are going to show implementation of multiple failure detectors, starting from a simple type and moving towards a more complicated and reliable type. We define a new namespace associated with everything related to the Node abstraction.

Implementing A Node

Our example distributed system consists of nodes. Each of the nodes can have neighbor nodes. Being a neighbors means knowing each other address, being able to communicate and track each other's health.

Let's define a module where we will store a Neighbor record type, consisting of a host and a port.

```
namespace Node

module DataTypes =

    type Neighbor = {
        host: string
        port: int
    }
```

The main abstraction we define to represent a node in a distributed system is a Node type.

When we want to start a node, we have several options to choose from. F# discriminated unions is a really useful feature that helps with this. For example, we define discriminated unions for NetworkProtocol, FailureDetectorType and ConsensusType to be able to indicate whether the node should use TCP or UDP, or certain failure detector or consensus types.

In the example distributed system, each node gets a unique host and port, as well as a value. Host and port will be used to work with a network server, and a value will be useful for consensus in later parts.

The defined discriminated unions are used in a NodeConfiguration type that is passed to the InitializeNode function of the Node type. NodeConfiguration type defines other node configuration fields, in addition to a network server, failure detector, and consensus algorithms that can be specified for a node. Each node can be started with a defined set of neighbors, apply given receiveMessageFunction to handle new user-defined message types that aren't service messages (e.g. not failure detection or consensus messages). We can also specify whether failure detectors should be using gossipping and whether we'd like to view verbose console logs.

The Node type itself has ReceiveMessage method, that is also passed into the network server as a processMessage function described in the previous part. InitializeNode method initializes node properties based on specified configuration, it initializes neighbors, failure detector, consensus, etc. Check out the full implementation in Node.fs.

```
namespace Node

type NetworkProtocol =
    | TCP
    | UDP

type FailureDetectorType =
    | PingAck
    | SimpleHeartbeat
    | HeartbeatRecovery
    | HeartbeatSlidingWindow
    | HeartbeatSuspectLevel
    | NoFailureDetector

type ConsensusType =
    | ChandraToueg
    | NoConsensus

type NodeConfiguration = {
    neighbors: Neighbor list
    networkProtocol: NetworkProtocol
    failureDetector: FailureDetectorType
    consensus: ConsensusType
    receiveMessageFunction: obj -> Async<unit>
    gossipping: bool
    verbose: bool
}

type Node (host, port, value) =

    // ...

    member x.InitializeNode(conf: NodeConfiguration) = async {
        // ...
    }

    member x.ReceiveMessage (message: byte []) = async {
        // ...
    }

    member x.DetectedFailure (neighbor: Neighbor) = async {
        // ...
    }

    /// Starts the server to listen for requests.
    member x.Start = async {
        // ...
    }

    member x.AddNewNeighbor neighbor = async {
        // ...
    }
    member x.UpdateValue newValue = async {
        // ...
    }

    member x.StartConsensus () = async {
        // ...
    }
```

As we have several types of failure detectors, we define a FailureDetectors module and general FailureDetector type, listing methods every failure detector should implement:

```
namespace Node

module FailureDetectors =

    [<AbstractClass>]
    type FailureDetector () =
        abstract member DetectFailures: Async<unit>
        abstract member ReceiveMessage: obj -> (Neighbor -> Async<unit>) -> Async<bool>
        abstract member AddNeighbor: Neighbor -> Async<unit>
        abstract member InitializeFailureDetector: NetworkServer -> HashSet<Neighbor> -> unit
        abstract member GetSuspectedList: Async<Neighbor list>
        abstract member AddSuspects: Neighbor list -> Async<unit>
```

Check out the full implementation in FailureDetectors.fs.

We are ready to implement failure detectors!

Ping-Ack Failure Detector

PingAck type implements a simple failure detector.

With PingAck, all nodes are sending Ping messages to their neighbors.

Whenever a node receives a Ping, it must respond to its neighbor with an Ack message.

The Ping and Ack message types are defined in the PingAckFailureDetector module.

```
namespace Node

module PingAckFailureDetector =

    type Ping = {
        messageId: string
        senderHost: string
        senderPort: int
    }

    type AckForMessage = {
        messageId: string
        inResponse: string
        senderHost: string
        senderPort: int
    }

    // ...
```

PingAckFailureDetector type implements the failure detector logic.

When a node doesn't receive an Ack from a neighbor on a Ping it previously sent in a reasonable time, it will consider the neighbor suspected to have failed. ReceiveMessage method indicates which methods will be handling which messages.

We define a value tolerateFailureFor to know what duration of absense of response to a Ping a node can tolerate, i.e. without adding the neighbor to the list of suspects.

Whenever this failure detector is started, it launches its DetectFailures workflow. Within the DetectFailures method, we start two nested workflows: ReportHealthWorkflow and DetectFailuresWorkflow.

ReportHealthWorkflow is responsible for a node sending out Ping messages to neighbors every pingInterval milliseconds.

DetectFailuresWorkflow is responsible for calculating whether the node received Ack messages from neighbors within reasonable time, and if not, marking the unresponsive neighbors as suspected. This workflow runs every failureDetectionInterval milliseconds.

For the full source code, check out FailureDetectorsPingAck.fs.

```
module PingAckFailureDetector =

    // ...

    type PingAckFailureDetector (host, port, verbose, detectedFailureFunction) =
        inherit FailureDetector ()

        // ...

        member val pingInterval = 4000
        member val failureDetectionInterval = 6000
        member val tolerateFailureFor = 10000L

        /// Collection of neighbors that are suspected to have failed.
        member val Suspected = new HashSet<Neighbor>()

        override x.InitializeFailureDetector (server: NetworkServer)

                                             (neighbors: HashSet<Neighbor>) =
            x.server <- server
            x.Neighbors <- neighbors
            x.NeighborsHealth <- new Dictionary<Neighbor, NodeHealthStatus>()
            for n in neighbors do x.NeighborsHealth.Add(n, NodeHealthStatus())

        override x.ReceiveMessage message updateNeighborsFunction = async {
            match message with
            | :? AckForMessage as ack ->
                do! x.HandleAck ack updateNeighborsFunction
                return true
            | :? Ping as ping ->
                do! x.HandlePing ping updateNeighborsFunction
                return true
            | _ -> return false
        }

        override x.DetectFailures = async {
            do! x.ReportHealthWorkflow |> Async.StartChild |> Async.Ignore

            do! x.DetectFailuresWorkflow |> Async.StartChild |> Async.Ignore
        }

        override x.AddNeighbor neighbor = async {
            // ...
        }

        member x.ReportHealthWorkflow = async {
            // ...
            while true do
                do! Async.Sleep x.pingInterval
                // ...
                for n in notSuspectedNeighbors do do! x.SendPing n
            // ...
        }

        member x.DetectFailuresWorkflow = async {
            printfn "Set up failure detection"
            try
                while true do
                    // Waiting failureDetectionInterval milliseconds
                    do! Async.Sleep x.failureDetectionInterval

                    // Detecting failure
                    for nh in x.NeighborsHealth do
                        let neighbor = nh.Key
                        let health = nh.Value

                        let howLongAckTook =

                            health.lastReceivedAckTime - health.lastSentPingTime

                        if Math.Abs howLongAckTook > x.tolerateFailureFor then
                            printfn "SUSPECTED FAILURE OF NEIGHBOR %s:%i" neighbor.host

                                                                          neighbor.port
                            x.Suspected.Add neighbor |> ignore
                            do! detectedFailureFunction neighbor
                        elif x.Suspected.Contains neighbor then
                            printfn "NEIGHBOR CAME BACK %s:%i" neighbor.host neighbor.port
                            x.Suspected.Remove neighbor |> ignore
            with
            | ex -> printfn "Detect Failures Workflow Exception: %s" ex.Message
        }

        // Nodes only send pings to neighbors
        member x.SendPing (neighbor: Neighbor) = async {
            // ...
            do! x.server.SendMessage pingMessage neighbor.host neighbor.port
            // ...
        }

        // Nodes only send acks to neighbors
        member x.SendAck (neighbor: Neighbor) (pingM: Ping) = async {
            // ...
            do! x.server.SendMessage ackMessage neighbor.host neighbor.port
            // ...
        }

        member x.HandlePing (ping: Ping) updateNeighborsFunction = async {
            // ...
            if x.Suspected.Contains neighbor then x.Suspected.Remove neighbor |> ignore
            do! x.SendAck neighbor ping
            // ...
        }

        member x.HandleAck (ack: AckForMessage) updateNeighborsFunction = async {
            // ...
        }
```

PingAck algorithm is a very simple, but not very powerful failure detector. Once a neighbor is marked as suspected, it never recovers.

Let's look at more algorithms.

Heartbeat Failure Detector

Heartbeats is another approach of implementing failure detectors. Heartbeat type defines a heartbeat.

```
namespace Node

module HeartbeatFailureDetector =

    type Heartbeat = {
        messageId: string
        senderHost: string
        senderPort: int
    }
```

Every heartbeatInterval milliseconds each node runs ReportHealthWorkflow to send out heartbeats to its neighbors. Every failureDetectionInterval milliseconds, each node runs DetectFailuresWorkflow to evaluate whether any of the neighbors have failed. During the evaluation, the algorithm checks how long did it take since the node received the last heartbeat from each neighbor. If the heartbeat roundtrip is longer than acceptable value (roundtripTime + heartbeatInterval), then the neighbor is suspected to have failed.

```
module HeartbeatFailureDetector =

    // ...

    type HeartbeatFailureDetector (host, port, verbose, detectedFailureFunction) =
        inherit FailureDetector ()

        // ...

        member val heartbeatInterval = 2000
        member val failureDetectionInterval = 4000

        /// Collection of neighbors that are suspected to have failed.
        member val Suspected = new HashSet<Neighbor>()

        override x.InitializeFailureDetector (server: NetworkServer)

                                             (neighbors: HashSet<Neighbor>) =
            x.server <- server
            x.Neighbors <- neighbors
            x.HeartbeatsInfo <- new Dictionary<Neighbor, HeartbeatInfo>()
            for n in neighbors do
                do x.HeartbeatsInfo.Add(n, HeartbeatInfo())

        override x.ReceiveMessage message updateNeighborsFunction = async {
            match message with
            | :? Heartbeat as heartbeat ->
                do! x.HandleHeartbeat heartbeat updateNeighborsFunction
                return true
            | _ ->
                return false
        }

        override x.DetectFailures = async {
            do! x.ReportHealthWorkflow |> Async.StartChild |> Async.Ignore

            do! x.DetectFailuresWorkflow |> Async.StartChild |> Async.Ignore
        }

        override x.AddNeighbor neighbor = async {
            // ...
        }

        member x.ReportHealthWorkflow = async {
            while true do
                // Waiting a number of milliseconds before sending heartbeats again
                do! Async.Sleep x.heartbeatInterval
                // ...
                for n in notSuspectedNeighbors do do! x.SendHeartbeat n
        }

        member x.DetectFailuresWorkflow = async {
            printfn "Set up failure detection."
            try
                while true do
                    // Waiting failureDetectionInterval milliseconds
                    do! Async.Sleep x.failureDetectionInterval

                    // Detecting failure
                    for heartbeatInfo in x.HeartbeatsInfo do
                        let neighbor = heartbeatInfo.Key

                        let lastHeartbeatTime= heartbeatInfo.Value.lastReceivedHeartbeatTime
                        let currentTime = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds()
                        let timeSinceLastHeartbeat = currentTime - lastHeartbeatTime

                        let acceptableHeartbeatRoundtripTime =
heartbeatInfo.Value.roundtripTime
                        let acceptableTimeSincePreviousHeartbeat = int64
(acceptableHeartbeatRoundtripTime + x.heartbeatInterval)

                        if timeSinceLastHeartbeat > acceptableTimeSincePreviousHeartbeat then
                            do! x.NeighborIsDown neighbor
                        elif x.Suspected.Contains neighbor then
                            do! x.NeighborCameBackUp neighbor
            with
            | ex->
                printfn "Detect Failures Workflow Exception: %s" ex.Message
        }

        member x.SendHeartbeat (neighbor: Neighbor) = async {
            // ...
            do! x.server.SendMessage heartbeatMessage neighbor.host neighbor.port
            // ...
        }

        member x.HandleHeartbeat (heartbeat: Heartbeat) updateNeighborsFunction = async {
```

```
        // ...
        if x.Suspected.Contains neighbor then do! x.NeighborCameBackUp neighbor
        // ...
    }

    member x.NeighborIsDown neighbor = async {
        printfn "SUSPECTED FAILURE OF NEIGHBOR %s:%i" neighbor.host neighbor.port
        x.Suspected.Add neighbor |> ignore
        do! detectedFailureFunction neighbor
    }

    member x.NeighborCameBackUp neighbor = async {
        printfn "NEIGHBOR %s:%i CAME BACK UP" neighbor.host neighbor.port
        x.Suspected.Remove neighbor |> ignore
    }
```

For the full source code, check out FailureDetectorsHeartbeat.fs.

This algorithm works a bit differently, and is able to remove recovered nodes from suspected list. This algorithm however, has a set value for acceptable roundtrip time. What happens if neighbors actually send heartbeats, but it takes consistently longer to deliver them, when our node thinks they have failed?

We are going to use and gradually improve this algorithm as a base model for next failure detectors.

Heartbeat Failure Detector With Adjustable Timeout

The HeartbeatRecoveryFailureDetector is very similar to HeartbeatFailureDetector in its operation.

Take a look at FailureDetectorsHeartbeatRecovery.fs for full implementation, and compare it to FailureDetectorsHeartbeat.fs if you'd like.

In HeartbeatRecoveryFailureDetector the heartbeat roundtripTime is flexible and can be adjusted, in distiction to what it was implemented like in the previous algorithm. Whenever the failure detector encounters a node reappearing after a suspected failure, it gives the roundtripTime a new value for acceptable heartbeat roundtrip based on how long it took a node to actually send a heartbeat.

```
type HeartbeatRecoveryFailureDetector (host, port, verbose, detectedFailureFunction) =
    inherit FailureDetector ()

    // ...

    /// Collection of neighbors that are suspected to have failed.
    member val Suspected = new Dictionary<Neighbor, LastReceivedHeartbeatTime>()

    // ...

    member x.NeighborCameBackUp neighbor = async {
        // New roundtrip time will be longer

        // based on how long it took the recovered node to come back

            let newRoundtripTime = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds() -
x.HeartbeatsInfo.[neighbor].lastReceivedHeartbeatTime
```

```
            x.Suspected.Remove neighbor |> ignore

            printfn "NEIGHBOR %s:%i CAME BACK UP. OLD ROUNDTRIP TIME: %i. NEW ROUNDTRIP
TIME: %i" neighbor.host neighbor.port x.HeartbeatsInfo.[neighbor].roundtripTime
newRoundtripTime

            x.HeartbeatsInfo.[neighbor].UpdateRoundtripTime(newRoundtripTime)
        }
```

The HeartbeatRecoveryFailureDetector is better than the simple HeartbeatFailureDetector because it can adjust the acceptable roundtrip. It's still not ideal. There can be a situation where a neighbor takes an unusually long time to send a heartbeat. This will set the acceptable roundtrip to an unusually large value. It will not help detecting future failures that happen earlier than the acceptable roundtrip time, because the failure detector will think that it's waiting for a response within the range of the acceptable, large heartbeat roundtrip time.

How can we help the failure detector find the reasonable roundrip time?

Heartbeat Failure Detector With Sliding Window

The HeartbeatSlidingWindowFailureDetector doesn't rely on a single roundtripTime value for an acceptable heartbeat roundtrip. It is more effective, because it keeps track of last slidingWindowSize number of heartbeat roundtrip values and calculates the AcceptableRoundtripTime based on the average heartbeat rountrip duration of the last slidingWindowSize heartbeats. This technique helps the failure detector to avoid the unreasonably high acceptable heartbeat roundtrip value, because the sliding window is always moving and the algorithm will adjust to the relevant acceptable heartbeat roundtrip.

Take a look at HeartbeatSlidingWindowFailureDetector.fs for full implementation, and compare it to FailureDetectorsHeartbeatRecovery.fs if you'd like.

```
type HeartbeatInfo() =

    let startingRoundtripTime = 2000L
    let slidingWindowSize = 50

    let averageRoundtripDuration (roundtrips: seq<LastReceivedHeartbeatTime>) =
        Seq.sum roundtrips / int64 (Seq.length roundtrips)

    member val roundtripTimes: LastReceivedHeartbeatTime list =
        [startingRoundtripTime] with get, set

    member x.AddRoundtrip (time: LastReceivedHeartbeatTime) =
        x.roundtripTimes <- time::x.roundtripTimes
        printfn "ADDED NEW ROUNDTRIP! NOW AVERAGE IS: %i" (x.AcceptableRoundtripTime ())

    member x.AcceptableRoundtripTime () =
        if Seq.length x.roundtripTimes <= slidingWindowSize then
            x.roundtripTimes |> averageRoundtripDuration
        else
            Seq.windowed slidingWindowSize x.roundtripTimes
```

```
                |> Seq.head
                |> averageRoundtripDuration

        member val lastReceivedHeartbeatTime =
            DateTimeOffset.UtcNow.ToUnixTimeMilliseconds() with get, set

        member x.UpdateLastReceivedHeartbeatTime time =
            x.lastReceivedHeartbeatTime <- time

    // ...

    type HeartbeatSlidingWindowFailureDetector (host, port, verbose, detectedFailureFunction)
=
        inherit FailureDetector ()

        // ...

        /// Collection of neighbors that are suspected to have failed.
        member val Suspected = new Dictionary<Neighbor, LastReceivedHeartbeatTime>()

        // ...

        member x.HandleHeartbeat (heartbeat: Heartbeat) updateNeighborsFunction = async {

            // ...
            let heartbeatRoundtrip = heartbeatReceivedTime -
x.HeartbeatsInfo.[neighbor].lastReceivedHeartbeatTime
            x.HeartbeatsInfo.[neighbor].AddRoundtrip heartbeatRoundtrip
            x.HeartbeatsInfo.[neighbor].UpdateLastReceivedHeartbeatTime heartbeatReceivedTime
            // ...
        }

        // ...

        member x.NeighborCameBackUp neighbor heartbeatReceivedTime = async {
            // New roundtrip time will be longer
            // based on how long it took the recovered node to come back.
            let heartbeatRoundtrip = heartbeatReceivedTime - x.Suspected.[neighbor]
            x.Suspected.Remove neighbor |> ignore
            x.HeartbeatsInfo.[neighbor].AddRoundtrip heartbeatRoundtrip
        }
```

This algorithm is rather effective. Let's take a look at another approach.

Heartbeat Failure Detector With Suspect Level

HeartbeatSuspectLevelFailureDetector is a slight modification of a HeartbeatSlidingWindowFailureDetector with changes in how failures are detected. In the previous failure detectors, nodes are suspected when actual heartbeat roundtrip exceeds the acceptable roundtrip time, and the algorithm for determining the acceptable roundtrip time valries.

With HeartbeatSuspectLevelFailureDetector, each theoretically failed node gets assigned a SuspectLevel based on how many heartbeats were lost. Whenever we receive a heartbeat from a theoretically suspected node, its SuspectLevel value is decremented. After a node's SuspectLevel reaches a value larger than suspectLevelMaximum, it is added to a failure detector's suspected list.

This apprpach allows for a more even suspicion process, tracking neighbors tend to fail more than others.

```
    type HeartbeatInfo() =

        // ...

        let mutable suspectLevel = 0

        member x.SuspectLevel
            // ...

        member x.ReduceSuspicion () =
            if x.SuspectLevel > 0 then x.SuspectLevel <- x.SuspectLevel - 1


    type HeartbeatSuspectLevelFailureDetector (host, port, verbose, detectedFailureFunction) =
        inherit FailureDetector ()

        // ...

        member val suspectLevelMaximum = 3

        /// Collection of neighbors that are suspected to have failed.
        member val Suspected = new Dictionary<Neighbor, LastReceivedHeartbeatTime>()

        // ...

        member x.DetectFailuresWorkflow = async {
            if verbose then printfn "Set up failure detection."
            while true do
                try
                    // Waiting failureDetectionInterval milliseconds
                    do! Async.Sleep x.failureDetectionInterval

                    // Detecting failure
                    for heartbeatInfo in x.HeartbeatsInfo do
                        let neighbor = heartbeatInfo.Key
                        if not <| x.Suspected.ContainsKey neighbor then
                            let lastHeartbeatTime =
heartbeatInfo.Value.lastReceivedHeartbeatTime
                            let currentTime = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds()
                            let timeSinceLastHeartbeat = currentTime - lastHeartbeatTime

                            let numberOfLostHeartbeats = int (timeSinceLastHeartbeat /
heartbeatInfo.Value.AcceptableRoundtripTime ())

                            if numberOfLostHeartbeats > 0 then
                                x.HeartbeatsInfo.[neighbor].SuspectLevel <-
numberOfLostHeartbeats
                                if numberOfLostHeartbeats >= x.suspectLevelMaximum then
                                    do! x.NeighborIsDown neighbor
                                else
                                    if verbose then printfn "Neighbor %s:%i is suspected at
level '%i'." neighbor.host neighbor.port x.HeartbeatsInfo.[neighbor].SuspectLevel
                with
                | ex ->
                    if verbose then printf "Detect Failures Workflow Exception: %s" ex.Message
        }

        // ...
        member x.HandleHeartbeat (heartbeat: Heartbeat) updateNeighborsFunction = async {

            // ...

            x.HeartbeatsInfo.[neighbor].ReduceSuspicion()
        }

        // ...
```

Gossipping Failure Detector

In many distributed systems every node doesn't always communicate with every single other node. In distributed systems with large number of nodes it could be impractical. Without all to all communication, information about failures isn't complete. For strong accuracy, we can use gossipping and apply it to failure detectors to make sure information about failures is eventually communicated across all of the nodes.

GossippingFailureDetector type defines such failure detector. It can use any of the previously defined failure detectors, and includes additional functionality. Every gossipInterval it sends out the node's suspected list to each neighbor. Whenever a node receives a suspected list from a neighbor, it combines it with its own suspected list. This way every node eventually finds out about all the other failures in the system, even if it doesn't directly communicate with all of the nodes.

```
namespace Node

module GossippingFailureDetector =

    type SendSuspectedList = {
        suspectedList: Neighbor list
    }

    type GossippingFailureDetector (failureDetector) =
        inherit FailureDetector ()

        [<DefaultValue>] val mutable innerFailureDetector : FailureDetector

        // ...

        member val gossipInterval = 10000

        override x.InitializeFailureDetector (server: NetworkServer) (neighbors:
HashSet<Neighbor>) =
            x.innerFailureDetector <- failureDetector
            x.server <- server
            x.neighbors <- neighbors
            x.innerFailureDetector.InitializeFailureDetector server neighbors

        override x.DetectFailures = async {
            do! x.innerFailureDetector.DetectFailures |> Async.StartChild |> Async.Ignore

            do! x.GossippingSuspects |> Async.StartChild |> Async.Ignore
        }

        override x.ReceiveMessage message updateNeighborsFunction = async {
            let! messageReceived = x.innerFailureDetector.ReceiveMessage message
updateNeighborsFunction
            if messageReceived then return true
            else
                match message with
                | :? SendSuspectedList as suspectList ->
                    do! x.HandleReceivedSuspectList suspectList
                    return true
                | _ -> return false
        }

        override x.AddNeighbor neighbor = async {
            do! x.innerFailureDetector.AddNeighbor neighbor
        }

        override x.AddSuspects neighbors = async {
            do! x.innerFailureDetector.AddSuspects neighbors
        }

        override x.GetSuspectedList = async {
            return! x.innerFailureDetector.GetSuspectedList
        }

        member x.GossippingSuspects = async {
            printfn "Set up gossipping susepcts schedule."
            try
                while true do
                    // Waiting a number of milliseconds before gossipping again
                    do! Async.Sleep x.gossipInterval

                    let! suspectedList = x.GetSuspectedList

                    if not (suspectedList |> List.isEmpty) then
                        printfn "Sending Suspect List: %A" suspectedList

                        // Communicating suspects to neighbors
                        for n in x.neighbors do
                            do! x.SendSuspects n suspectedList
            with
            | ex-> printfn "Gossipping Suspects Workflow Exception: %s" ex.Message
        }

        member x.SendSuspects neighbor suspectedList = async {
            do! x.server.SendMessage { suspectedList = suspectedList } neighbor.host
neighbor.port
        }

        member x.HandleReceivedSuspectList suspectList = async {
            printfn "Received Suspect List: %A" suspectList.suspectedList
            do! x.AddSuspects suspectList.suspectedList
        }
```

Next

Learn more in the next part on implementing consensus using failure detectors.