Diving Into Failure Detectors and Consensus

With Examples in F#

Natallia Dzenisenka

@nata_dzen

2019

The Theory behind Failure Detectors

By Natallia Dzenisenka

Table Of Contents

Failure Detectors

According to a common definition, a distributed system is a group of computers working together that appears to the client as a single computer. Beyond this seemingly simple definition, there are some challenges to overcome to provide reliable operation. There are many things that happen behind the scenes that might violate the smooth functioning of a distributed system. A lot of the issues are caused by the distributed and asynchronous nature of the environment.

Imagine a system that stores data evenly distributed between nodes, where each node is responsible for a certain range of data. A client can send a request to the system, and it can land on any node. Let's call the node that received the request a *coordinator*. After receiving the request, the coordinator's mission is to fulfill it, either alone, or working together with other nodes. In our example system, each node knows what range of data it and other nodes are responsible for storing. This way when a coordinator receives a request, it already knows which node to contact to provide the response.
If our distributed system stored data about grocery products, the client could decide to request more information about the product with the name "Sweet Tea" produced by a certain manufacturer. The coordinator would look at the key parts of data in the client's request, and figure out that all data about tea products and this specific manufacturer is the responsibility of some node. Let's call it the *main data node* in this example. The coordinator will ask the main data node to provide information about what the client requested. In the perfect world, the main data node will provide a reply, and the coordinator will successfully return it to the client. In reality, there might be issues with the availability of nodes in the distributed system, such as *node crashes* or *failures*. If the system isn't redundant, and the coordinator doesn't have a backup plan and tries to get information from the crashed node, it will likely have wait until the crashed node comes back up. This might take a very long time, and most clients won't be thrilled to wait forever.

Our systems won't be responsive and reliable if they don't know how to detect and get around possible failures that are common in the real world.

There is a need for a mechanism that would help each node in the system to detect when crucial components are not working. The client doesn't necessarily care whether the main data node is up or down, it still expects the answer from the distributed system.

Every system is different, but in most of them, we need to be ready to face failures and have a plan on dealing with failures. To do that, we need to understand different kinds of failures.

Failures Are Everywhere

Ignoring a chance of failure instead of facing the reality of it is one of the mistakes we can avoid when designing a new distributed system. Thinking that a specific type of failure isn't likely to happen is a bold assumption that needs to be thoroughly evaluated. Some types of failures are less frequent than others. Some of them are rare but can lead to greater damage in your system.

A system can encounter *value failures* and *timing failures*. Failures can be consistent and *inconsistent*, *persistent* and *intermittent*. There are many ways to classify failures in a distributed system.

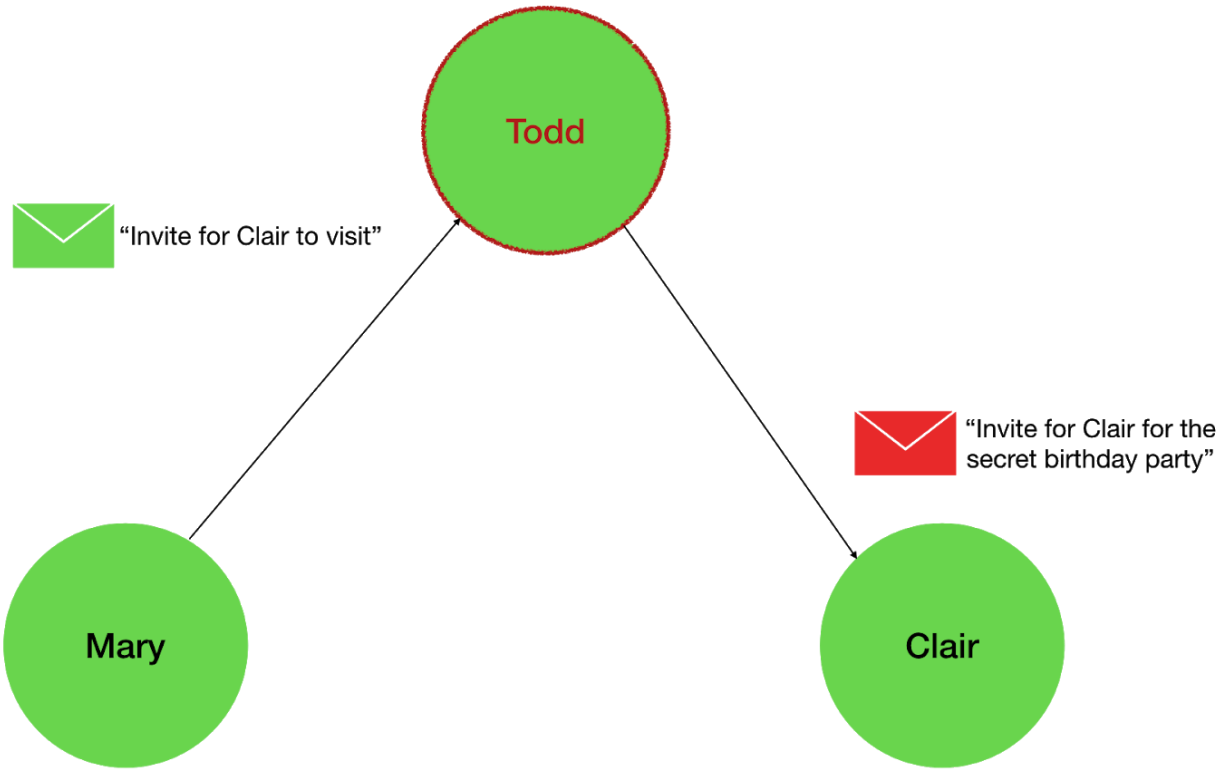
Value failures happen when a node intentionally sends incorrect information in reply to another node. Imagine Mary sends a message to Todd asking him to invite their best friend Clair for a visit. Mary also warns him not to tell Clair it's about the surprise party for her birthday. Then imagine, Todd replies to Mary confirming the promise to keep the party a surprise and immediately calls Clair to reveal the secret. Todd's wrong behavior is a *value failure*. This isn't what Mary expected from Todd, and Mary still thinks Clair doesn't know anything about the surprise.

Timing failures happen when a node doesn't provide a response within the expected timeframe. A timing failure could happen if Mary decided to order a cake for Clair's birthday, and haven't received it after two weeks of waiting, even though the bakery promised to deliver it within a two week period.
A failure is consistent when every node experiences the failure the same way. A failure is inconsistent when different nodes perceive the failure differently, for example, one node may experience it, and another node may have no issues.
A *failure mode* is a variation in which a distributed system can fail. Let's cover several failure modes we commonly work with.

Performance Failures happen when the nodes don't experience *value failures* but have *timing failures*, and are delivering responses outside the time frame expected by the system. A performance failure would happen if Todd kept a promise he gave to Mary to invite Clair for a visit, but forgot to actually do that on time and Clair missed the surprise birthday party.

Byzantine Failures occur when some nodes misbehave by intentionally stating that the correct value is incorrect, or vice versa. For example, a node having a Byzantine failure may point other nodes to the false leader. Todd was affected by a Byzantine failure when he was intentionally passing the wrong information to Clair.

Diagram - Byzantine Failure Example



Authenticated Byzantine Failures are similar to Byzantine Failures, with one distinction. Here a node can't forge messages it received from others, whereas with regular Byzantine Failures it can. If Todd was affected by an Authenticated Byzantine Failure instead of general Byzantine Failure, he wouldn't be able to corrupt the requests he received from Mary or anyone else and pass the incorrect message to Clair. He would still be able to pass wrong and misleading information to his friends as long as he himself is the source of the information.

Omission Failures result from one or more failed attempts when communicating with a node. Failure to send or receive messages is an omission failure. An omission failure could happen when the bakery actually sent the cake to Mary, but it got lost or eaten by the hungry delivery person on the way from the bakery to Mary's house.

Crash Failures happen after all attempted interactions with a node result in omission failures. If a node fails by stopping because of power outages, software or hardware errors, or other reasons, and other nodes don't find out of the fail - it's a crash failure. In other words, there a crash failure when a node completely fails to respond. If the bakery building caught on fire and Mary didn't know about it, that

would be a crash failure which would explain why Mary didn't receive the cake from the bakery, and every single attempt to contact the bakery to ask about the cake, or order a new cake failed.

Fail-Stop Failures are similar to Crash Failures, with a difference that all correctly-functioning nodes can detect and learn about the failure of the specific failed node.

Discovering Failures

Failure detectors are one of the essential techniques to discover *node crashes or failures* in a distributed system. It helps processes in a distributed system to change their action plan when they face such failures.

For example, failure detectors can help the coordinator node avoid the unsuccessful attempt of requesting data from a node that crashed. With failure detectors, each node will know if any other nodes in the cluster crashed. Having this useful information, each node has the power to decide what to do in case of the detected failures of other nodes. For example, instead of contacting the main data node, the coordinator node could decide to contact one of the healthy secondary replicas of the main data node and provide the response to the client.

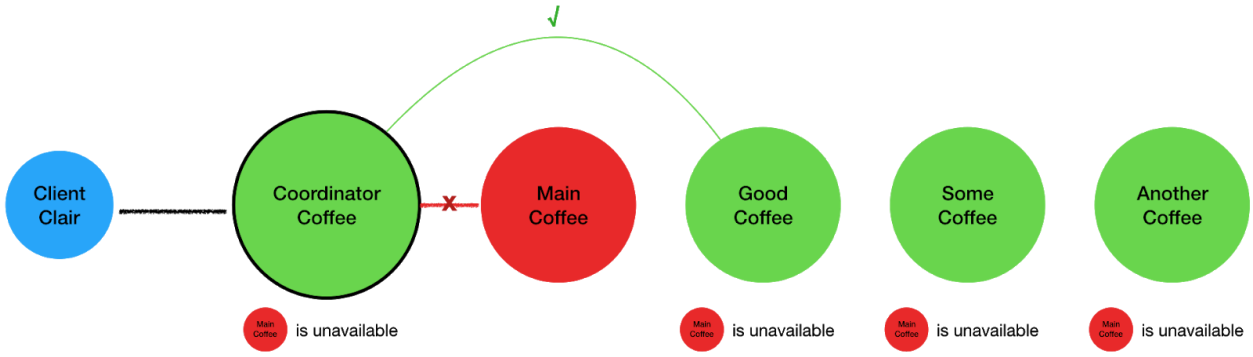
Failure detectors don't guarantee the successful execution of client requests. Failure detectors are important, as they help nodes in the system to be aware of known crashes of other nodes and avoid continuing the path of failure. Failure detectors collect and provide information about node failures, and it's up to the distributed system logic to decide how to use it. If the data is stored redundantly across several nodes, the coordinator can choose to contact alternative nodes to execute the request. In other cases, there might be failures that could affect all replicas, then the client request isn't guaranteed to succeed.

Failure Detectors Of The Real World

To draw the parallel with the real world, imagine a network of coffee supply locations, where each store carries and sells a rotating variety of coffee beans in large amounts. You can think of each coffee location as a *node* in the distributed system. Each supply location stores coffee beans in smart containers with sensors that track the bean levels to tell how much coffee is left for each type of coffee. Supply locations continuously exchange this information and every coffee store knows the status of all the other stores.

One day, Clair decided to stop by one of the coffee locations called *Coordinator Coffee*. You can think of it as a *coordinator* node in a distributed system. Clair wanted to get some of her favorite coffee beans called the *best coffee beans*. The store assistant at *Coordinator Coffee* found that this particular supply location doesn't carry the *best coffee beans* anymore, but the *Main Best Coffee* is now the main location Clair could get her favorite coffee beans at. What Clair doesn't know about is that *Main Best Coffee* store location had a sudden power outage and had to close early for the day. Good thing all the coffee locations are exchanging data about their status, which acts as a coffee store *failure detector*. Because_ Main Best Coffee_ stopped sending the coffee sensor data, all the other locations including the *Coordinator Coffee* started suspecting that *Main Best Coffee* might be unavailable. Based on the data from the coffee failure detection system, the *Coordinator Coffee* assistant recommended Clair should skip the *Main Best Coffee* and go straight to another Good Coffee store that carries the *best coffee beans* and is open.

Diagram - Failure Detection Of The Real World



The mechanism for making each distributed system node aware of other node crashes is a failure detector. Failure detectors discover node crashes and help improve the reliability of the distributed system. After the system is aware of the discovered crashes, it will aim to get around them to avoid further failures.

Problems Solved With Failure Detectors

Many distributed algorithms rely on failure detectors. Even though failure detectors can be implemented as an independent component and used for things like reliable request routing, failure detectors are widely used internally for solving agreement problems, consensus, leader election, atomic broadcast, group membership, and other distributed algorithms.

Failure detectors are substantial for consensus and can be applied to improve reliability and help distinguish between nodes that have delays in their responses and those that crashed. Consensus algorithms can benefit from using *failure detectors* that *estimate* which nodes have crashed, even when the estimate isn't a hundred percent correct.

Failure detectors can improve atomic broadcast, the algorithm that makes sure messages are received in the same order by every node in a distributed system. It's common for atomic broadcast to use group membership algorithms to overcome failures. Unreliable failure detectors can be a significant optimization for the atomic broadcast in certain environments. A mistake made by the group membership algorithm could have a greater resulting cost than a mistake made by a failure detector. When a group membership algorithm detects a node crash, it needs to remove the failed node from the network to make sure the system isn't blocked. Often there's a need to replace the crashed node with a new one. These two are expensive operations in a distributed system. Unreliable failure detectors can be more efficient for atomic broadcast than group membership when incorrect *failure suspicions* (studied in the next section) are common. However, systems with an unlimited number of faulty nodes still prefer to rely on group membership algorithms instead of failure detectors for greater reliability.

Failure detectors can also be used to implement group membership algorithms, detectors, and in k-set agreement in asynchronous dynamic networks.

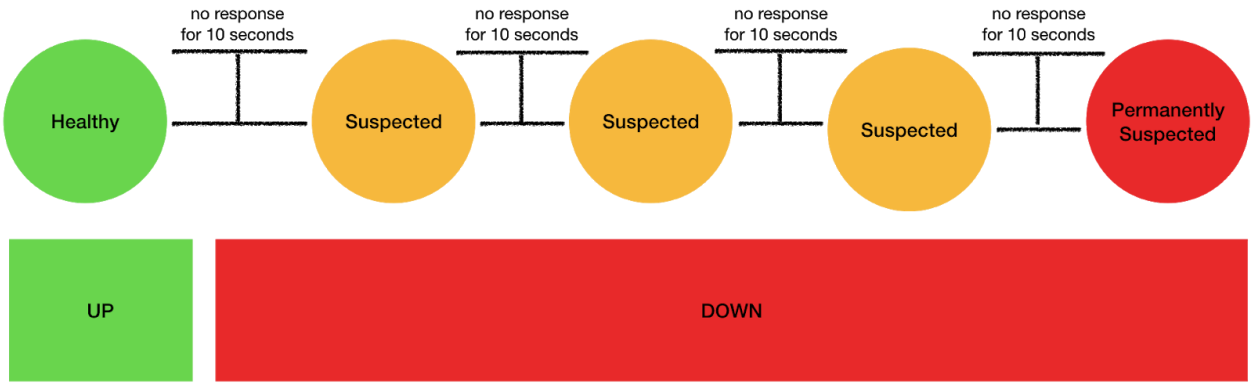
Properties Of A Failure Detector

In the account of differences in the nature of environments our systems run in, there are several types of failure detectors we can use. What makes them different?

In a synchronous system, a node can always determine if another node is up or down because there's no nondeterministic delay in message processing.

In an asynchronous system, we can't make an immediate conclusion that a certain node is down if we didn't hear from it. What we can do is start *suspecting* it's down. This gives the suspected failed node a chance to prove that it's up and didn't actually fail, just taking a bit longer than usual to respond. After we gave the suspected node enough chances to reappear, we can start *permanently suspecting it*, making the conclusion that the target node is down.

Diagram - Suspecting Failures



Can we discover *all* the failures? How precise can we be in our failure suspicions?

The answers to these questions can be different, which is why there are different types of failure detectors offering different guarantees.

Completeness is a property of a failure detector that helps measure whether some or all of the correctly-functioning nodes discovered each of the failed nodes. What are possible variations of completeness when working with failure detectors? How do we measure whether a failure detector has a *strong* or *weak* completeness property?

When every correctly-operating node *eventually* discovers every single failed node, it indicates that the failure detector has strong completeness. Eventually, because a node can first suspect another node to have possibly failed for a while, before marking it as permanently suspected.

Strong completeness property means that every node will eventually permanently suspect all the failed nodes.

Weak completeness means that some nodes will eventually permanently suspect all the failed nodes.

Diagram - Strong Completeness

a. All nodes are healthy.
b. Nodes 2 and 3 crashed.
c. The rest of the nodes eventually notice all crashed nodes.

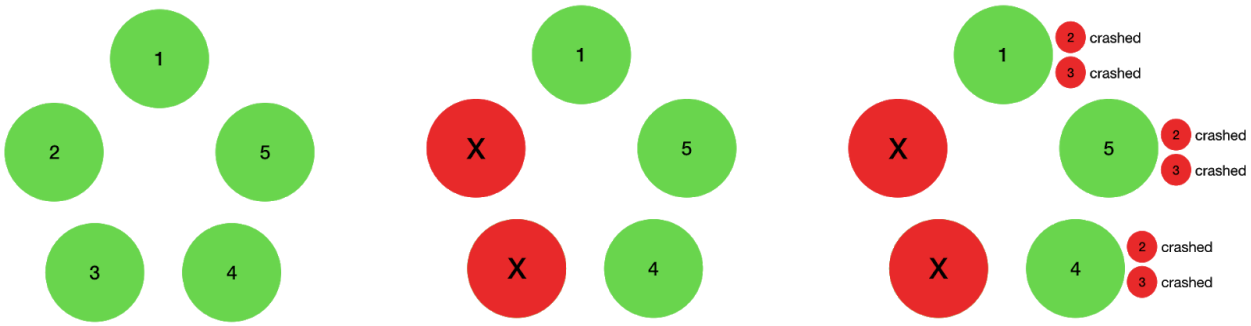
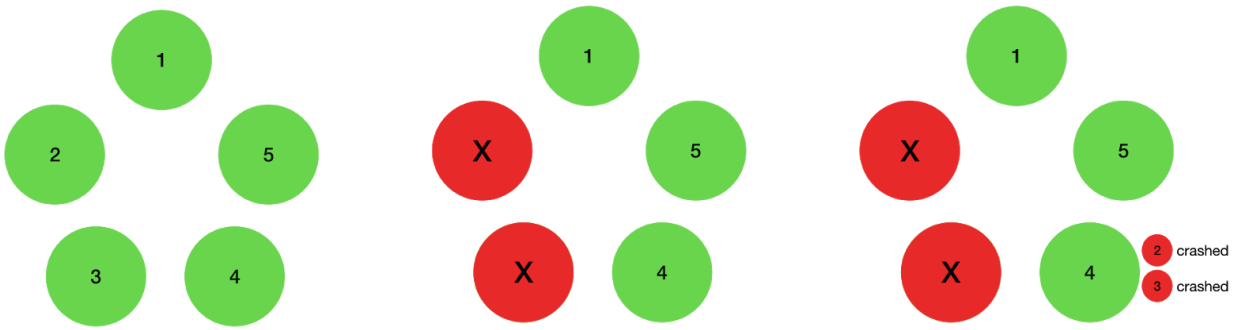


Diagram - Weak Completeness

a. All nodes are healthy.
b. Nodes 2 and 3 crashed.
c. Node 4 eventually discovers all the crashes.



Note that it's not enough for a failure detector to be strongly complete to be effective. By definition, a failure detector that immediately suspects every node will have strong completeness, but being oversuspicious doesn't make it useful in practice.

We need to have a way to measure whether the suspicion is true or false. Can a node start suspecting another node that hasn't crashed yet?

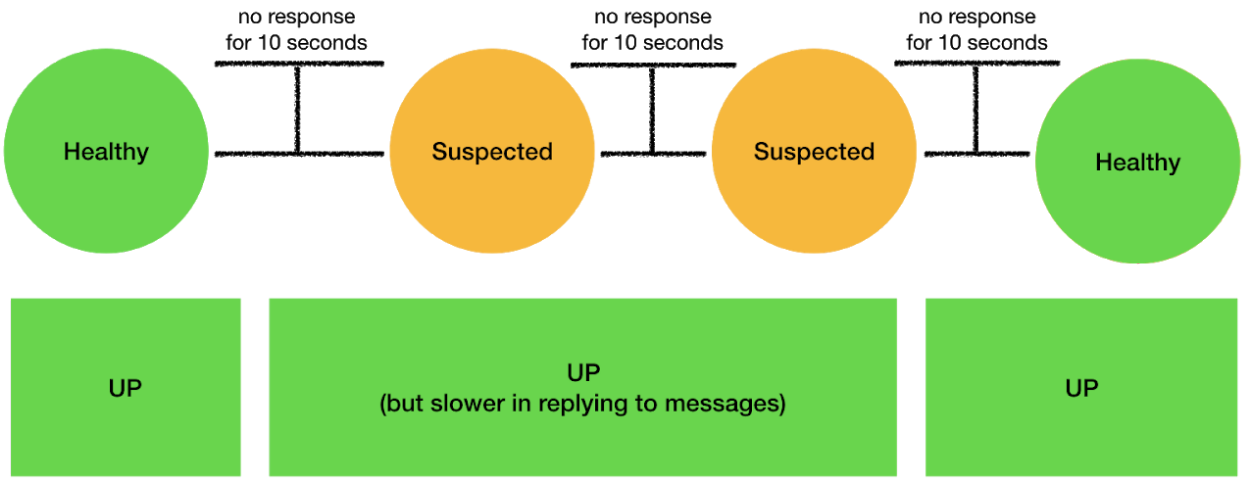
Accuracy is another important property of a failure detector. It shows how precise our suspicions are. In some situations, we might be wrong by incorrectly suspecting that a node is down when, in reality, it is up. Accuracy property defines how big a mistake from a failure detector is acceptable in suspecting the failure of another node.

With *strong accuracy*, no node should ever be suspected in failure by anyone before its actual crash. *Weak accuracy* means that some nodes are never incorrectly suspected by anyone. In reality, is hard to guarantee both strong and weak accuracy. They both don't allow for a failure detector to ever be

mistaken in its suspicion. Therefore, weak accuracy might sound misleading. For example, there could be a scenario where a node is suspecting another node in failure but stopped suspecting it after a very short while. This scenario is pretty common in reality, however, it doesn't satisfy even weak accuracy. Even with weak accuracy, an incorrect suspicion isn't allowed.

Strong and weak accuracy properties are sometimes called *perpetual accuracy*. Failure detectors with perpetual accuracy must be precise in their suspicion from the very first attempt.

Diagram - Eventual, Not Perpetual Accuracy



Because the failure detector starts suspecting the failure of a node that in reality up, and later changes its suspicion - this scenario shows eventual accuracy. Failure detectors with perpetual accuracy won't allow this scenario.

To allow a failure detector to change its mind and forgive temporarily wrong suspicions, there are two additional kinds of accuracy - *eventually strong* and *eventually weak*. In distinction to perpetual accuracy, failure detectors with eventual accuracy are allowed to initially be mistaken in their suspicions. But eventually, after some period of confusion, they should start behaving like failure detectors with strong or weak accuracy properties.

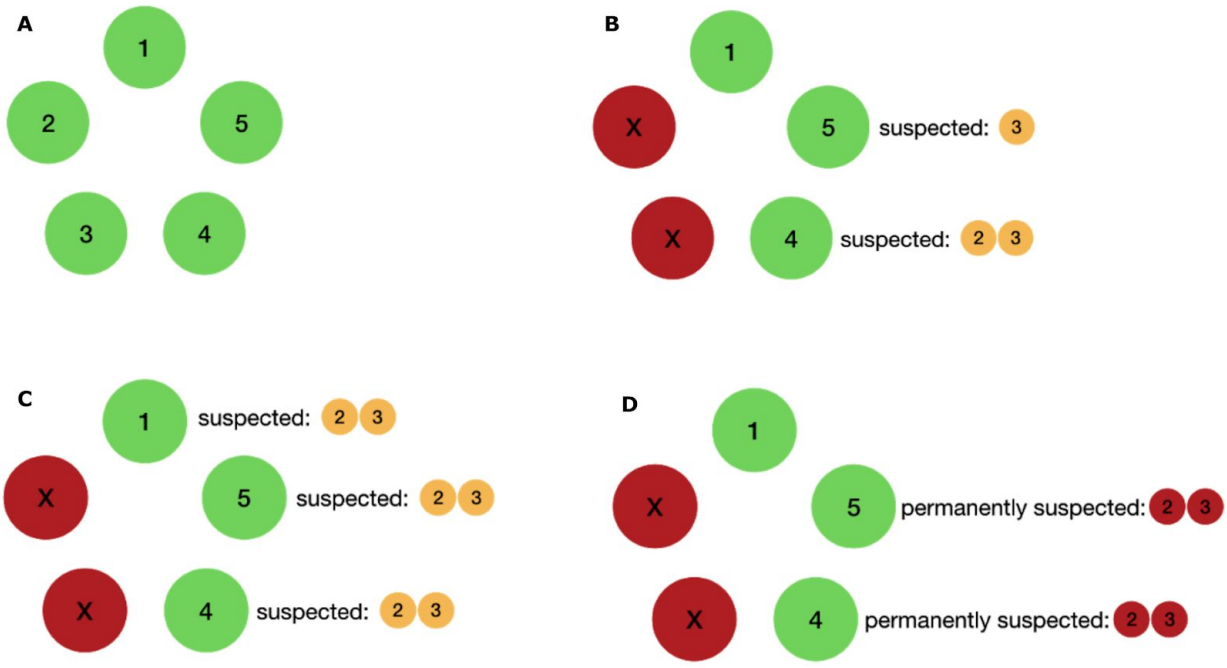
Eventually strong accuracy means a healthy node should be never suspected by anyone, after a possible period of uncertainty.

Eventually week accuracy means that some healthy nodes should be never suspected by anyone, after a possible period of uncertainty.

Diagram - Eventually Strong Accuracy

a. All nodes are healthy.
b. 10 seconds later: nodes 2 and 3 are down and are suspected by the rest of the nodes.
c. 10 more seconds later: nodes 2 and 3 still down and are suspected by the rest of the nodes.
d. 10 more seconds later: nodes 2 and 3 are still down and are finally permanently suspected by the rest of the nodes.

Diagram - Eventually Weak Accuracy



a. All nodes are healthy.
b. 10 seconds later: nodes 2 and 3 are down and are suspected by some of the nodes.
c. 10 more seconds later: nodes 2 and 3 still down and are suspected by some of the nodes.
d. 10 more seconds later: nodes 2 and 3 are still down and are finally permanently suspected by some of the nodes.

Types Of Failure Detectors

It is important to thoroughly understand the concepts of completeness and accuracy, as they are the foundation for comprehending classes of failure detectors. Each of the *failure detector* types embodies a combination of specific completeness and accuracy properties, resulting in eight types of failure detectors by T.D. Chandra and S. Toueg classification.

1. Perfect Failure Detector
   Strong Completeness, Strong Accuracy

2. Eventually Perfect Failure Detector:
   Strong Completeness, Eventual Strong Accuracy

3. Strong Failure Detector
   Strong Completeness, Weak Accuracy

4. Eventually Strong Failure Detector
   Strong Completeness, Eventual Weak Accuracy

5. Weak Failure Detector
   Weak Completeness, Weak Accuracy

6. Eventually Weak Failure Detector
   Weak Completeness, Eventual Weak Accuracy

7. Quasi-Perfect Failure Detector
   Weak Completeness, Strong Accuracy

8. Eventually Quasi-Perfect Failure Detector
   Weak Completeness, Eventual Strong Accuracy

Failure Detectors In Asynchronous Environment

Each of the failure detector types can be useful in different environments. In a fully asynchronous system, we can't make any timing assumptions. There is no time bound on when a message should arrive. In reality, we are often dealing with something in between an asynchronous and a synchronous system. We might not know how long it's going to take for a message to arrive, but we know it should eventually be delivered. We just don't know when.

Some distributed system problems are proved to be impossible to solve in a fully asynchronous environment in the presence of failures. For example, the *impossibility results* (FLP) - showed that it's impossible to solve consensus or atomic broadcast in an asynchronous system with even one crash failure. In such an environment, we can't reliably tell if a node is down, or just takes longer to respond. *But how do failure detectors apply to the impossibility results?*

To provide an option to overcome the impossibility results, T.D. Chandra and S. Toueg showed that it is indeed possible to solve consensus in an asynchronous system by adding a failure detection mechanism. T.D. Chandra and S. Toueg introduced the types of failure detectors, showed that some of the types can be transformed into others. Not only T.D. Chandra and S. Toueg created a classification model, but they also described failure detector applications for solving consensus and showed which failure detectors are minimally sufficient for solving consensus problems. T.D. Chandra and S. Toueg proved that *Eventually Weak Failure Detector* is the minimally sufficient type that can be used to solve the problem of consensus in an asynchronous system with the majority of correct nodes, allowing for the number of failures f = N/2 - 1. They also proved that *Weak Failure Detector* is the minimally sufficient type that can be used for solving consensus in such a system with any number of failures f = N - 1. More about how failure detectors can be used to solve consensus is described in one of the next parts.

What does this mean for the distributed systems we are building?

How would we build a real failure detection mechanism?

In real-world systems, a lot of failure detectors are implemented using heartbeats and timeouts in practical distributed systems. For example, a node can be regularly sending a message to everyone sharing that it's alive. If another node doesn't receive a heartbeat message from its neighbor for longer than a specified timeout, it would add that node to its suspected list. In case the suspected node shows up and sends a heartbeat again, it will be removed from the suspected list.

What is the completeness and accuracy for this algorithm in the asynchronous environment?

The completeness and accuracy properties of this timeout based failure detector depend on the chosen *timeout value*. If the timeout is too low, nodes that are sometimes responding slower than the timeout will alternate between being added and removed from the suspected list infinitely many times. Then, the completeness property of such a failure detector is weak, and the accuracy property is even weaker than eventual weak accuracy.

Can we transform this failure detection algorithm to at least satisfy the Eventually Weak Failure Detector properties - weak completeness and eventual weak accuracy?

To achieve *weak accuracy* we need to find the timeout value after which a node will not receive false suspicions. We can do this by changing the timeout and increasing its value after each false suspicion. With this, there should be a timeout big enough for nodes to deliver their heartbeats in the future. In practice, this puts the described failure detector with an increasing timeout algorithm in the range of *Eventually Weak Failure Detectors*.

Reducing, or *transforming a failure detector* algorithm with one set of completeness and accuracy properties into a failure detector algorithm with another set of such properties means finding a *reduction* or *transformation algorithm* that can complement the original failure detection algorithm and guarantee that it will behave the same way as the target failure detection algorithm in the same environment given the same failure patterns. This concept is formally called *reducibility* of failure detectors.

We can say that the original failure detector algorithm based on timeouts (described earlier) was *reduced* or *transformed* into an *Eventually Weak Failure Detector* by using increasing timeouts. As T.D. Chandra and S. Toueg showed, turns out it is also possible to transform failure detectors with *weak completeness* into failure detectors with *strong completeness*.

Because in reality it can be difficult to implement strongly complete failure detectors in asynchronous systems, per T.D. Chandra and S. Toueg we can transform failure detectors with *weak completeness* class of into failure detectors with *strong completeness*. This concept is formally called reducibility of failure detectors. For example, we can say that a Weak Failure Detector can be reduced to a Strong Failure Detector, and so on.

Reducibility Of Failure Detectors

Weak Failure Detector	→	Strong Failure Detector
Eventually Weak Failure Detector	→	Eventually Strong Failure Detector
Quasi-Perfect Failure Detector	→	Perfect Failure Detector
Eventually Quasi-Perfect Failure Detector	→	Eventually Perfect Failure Detector

What exactly does it mean for a failure detector to be reducible to another failure detector?

On its own, the class of failure detectors with weak completeness has limited benefits because it doesn't guarantee for all failures to be eventually discovered. By reducing or transforming weaker completeness failure detectors into stronger completeness failure detectors we can solve a broader range of problems.

You might be thinking, why will a failure detector that doesn't discover all the failures and keeps making mistakes ever be useful for any scenario? It turns out, there are many cases where weaker failure detectors can still be useful in practice, with slight modifications. When a weakly complete failure detector is reduced to a strongly complete failure detector, it opens the door for us to solve trickier problems that require string completeness. For example, consensus, atomic broadcast, and similar distributed algorithms can only be solved with *Strong Failure Detectors* or *Eventually Strong Failure Detectors*. Thus, per T.D. Chandra and S. Toueg, it is also possible to use *Weak Failure Detectors* or *Eventually Weak Failure Detectors* to solve such problems in an asynchronous system, if we apply the concept of reducibility. A lot of agreement algorithms rely on having a leader, and weak failure detectors can play a great role in being able to choose a reliable leader.

Next

There are many great papers on the theory of failure detectors.

Take a look at Unreliable Failure Detectors for Reliable Distributed Systems.

Learn more in the next parts of this series on failure detector and consensus implementations.

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.

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.

Part IV: Implementation of Consensus

By Natallia Dzenisenka

Table Of Contents

Implementing Consensus Based On Failure Detectors

Chandra-Toueg Consensus

Chandra-Toueg consensus is based on strong or eventually stong failure detectors. It operates in asynchronous environment and can tolerate number of failures f equals to N/2 - 1 where N is overall number of nodes in a distributed system.

The algorithm operates in rounds. In each round, one of the nodes is designated as a Coordinator, computed by the formula R mod N, where R is round number. For example, in a system with three nodes, in the first round, second node will be a Coordinator (index one). In the second round, third node will be a Coordinator (index two). In the third round, first node will be a coordinator (index zero), etc.

When there are no Coordinator failures encountered, the algorithm terminates in a single round. When there's a failure of the Coordinator encountered, the algorithm proceeds to the next round, and chooses a new Coordinator until there is no Coordinator failure. The algorithm needs the following messages to operate:

```
namespace Node

module Consensus =

    type ConsensusPreference = {
        round: int
        preference: obj
        timestamp: int64
    }

    type ConsensusCoordinatorPreference = {
        round: int
        preference: obj
    }

    type ConsensusPositiveAcknowledgement = {
        round: int
    }

    type ConsensusNegativeAcknowledgement = {
        round: int
    }

    type ConsensusDecide = {
        preference: obj
    }

    type RequestConsensus = {
```

```
        round: int
    }
```

ConsensusPreference can be sent by each node and received by the Coordinator. When a node received ConsensusPreference message, it means others see it as the Coordinator. Coordinator will track the message and once it received ConsensusPreference messages from majority of nodes, it will choose the ConsensusPreference message with the latest timestamp, and send it as ConsensusCoordinatorPreference message to everyone.

When a node receives ConsensusCoordinatorPreference, it sends ConsensusPositiveAcknowledgement to the Coordinator, and sets Decision value to the Coordinator Preference.

When a node received ConsensusPositiveAcknowledgement message from a neighbor, it means that neighbor sees it as the Coordinator, and agrees to the preference previously sent by the Coordinator. The Coordinator waits to receive ConsensusPositiveAcknowledgement messages from the majority of neighbors, and sends ConsensusDecide message to everyone.

When a node receives a ConsensusDecide message, that means it's completely ready to set the value to the one that has been sent within the ConsensusDecide message.

If node's failure detector found a failure of some neighbor, the node checks if the crashed neighbor is a Coordinator. If so, it sends a ConsensusNegativeAcknowledgement to the Coordinator, increases the round number and proposes the value to the new Coordinator.

```
module Consensus =

    // ...

  type ChandraTouegConsensus (server: NetworkServer, node, neighbors, initialValue) =

        // ...

        member val Value = initialValue with get, set
        member val Round = 0 with get, set
        member val Decision = initialValue with get, set
        member val ReceivedPreference = // ...
        member val ReceivedPositiveAcknowledgement = // ...
        member val ReceivedNegativeAcknowledgement = // ...

        member x.GetCoordinator round =
            let coordinatorIndex = round % (x.Neighbors.Count + 1)
            let orderedNodes = Seq.sort (x.Node::List.ofSeq x.Neighbors)
            let coordinator = orderedNodes |> Seq.item coordinatorIndex
            coordinator

    member x.ReceiveMessage (message: obj) = async {
            match message with
            | :? ConsensusPreference as preference ->
                do! x.HandlePreference preference
                return true
            | :? ConsensusCoordinatorPreference as coordinatorPreference ->
                do! x.HandleCoordinatorPreference coordinatorPreference
```

```
            return true
        | :? ConsensusPositiveAcknowledgement as ack ->
            do! x.HandlePositiveAcknowledgement ack
            return true
        | :? ConsensusNegativeAcknowledgement as nack ->
            do! x.HandleNegativeAcknowledgement nack
            return true
        | :? ConsensusDecide as decision ->
            do! x.HandleDecide decision
            return true
        | :? RequestConsensus ->
            do! x.StartConsensus ()
            return true
        | _ -> return false
    }

member x.StartConsensus () = async {
        x.Round <- x.Round + 1

        let proposal: ConsensusPreference =
            {
                round = x.Round
                preference = x.Value
                timestamp = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds()
            }

        let newCoordinator = x.GetCoordinator x.Round

        if newCoordinator = x.Node then do! x.HandlePreference proposal
        else do! x.server.SendMessage proposal newCoordinator.host

                                                newCoordinator.port
    }

    member x.HandlePreference (preference: ConsensusPreference) = async {
        if x.ReceivedPreference.ContainsKey preference.round then
            x.ReceivedPreference.[preference.round] <-

                 preference::x.ReceivedPreference.[preference.round]
        else x.ReceivedPreference.[preference.round] <- [preference]

        if x.ReceivedPreference.[preference.round].Length >=

           ((x.Neighbors.Count + 1) / 2) + 1 then
            let latestTimestampPreference =
                x.ReceivedPreference.[preference.round]
                |> List.maxBy (fun p -> p.timestamp)
            let coordinatorPreference: ConsensusCoordinatorPreference =
                {round = latestTimestampPreference.round;

                 preference = latestTimestampPreference.preference}
            for n in x.Neighbors do
                do! x.server.SendMessage coordinatorPreference n.host n.port
            do! x.HandleCoordinatorPreference coordinatorPreference
    }
```

```
member x.HandleCoordinatorPreference

    (coordinatorPreference: ConsensusCoordinatorPreference) = async {
    let ack: ConsensusPositiveAcknowledgement =

        {round = coordinatorPreference.round}
    let coordinator = x.GetCoordinator coordinatorPreference.round
    if coordinator = x.Node then do! x.HandlePositiveAcknowledgement ack
    else do! x.server.SendMessage ack coordinator.host coordinator.port
    x.Decision <- coordinatorPreference.preference
}

member x.HandlePositiveAcknowledgement

       (ack: ConsensusPositiveAcknowledgement) = async {
    if x.ReceivedPositiveAcknowledgement.ContainsKey ack.round

    then x.ReceivedPositiveAcknowledgement.[ack.round] <-

             ack::x.ReceivedPositiveAcknowledgement.[ack.round]
    else
        x.ReceivedPositiveAcknowledgement.[ack.round] <- [ack]

    if x.ReceivedPositiveAcknowledgement.[ack.round].Length >=

        ((x.Neighbors.Count + 1) / 2) + 1

    then
        let latestTimestampPreference =

        x.ReceivedPreference.[ack.round] |> List.maxBy (fun p -> p.timestamp)
        let decide: ConsensusDecide =

            { preference = latestTimestampPreference }
        for n in x.Neighbors do do! x.server.SendMessage decide n.host n.port
        do! x.HandleDecide decide
}

member x.HandleNegativeAcknowledgement

    (nack: ConsensusNegativeAcknowledgement) = async {
    if x.ReceivedNegativeAcknowledgement.ContainsKey nack.round

    then
        x.ReceivedNegativeAcknowledgement.[nack.round] <-

           nack::x.ReceivedNegativeAcknowledgement.[nack.round]
    else
        x.ReceivedNegativeAcknowledgement.[nack.round] <- [nack]

    if x.ReceivedNegativeAcknowledgement.[nack.round].Length >=

       ((x.Neighbors.Count + 1) / 2) + 1

    then do! x.ClearState ()
}
```

```
        member x.HandleDecide (decision: ConsensusDecide) = async {
            x.Value <- decision.preference
            do! x.ClearState ()
        }

        member x.DetectedFailure (neighbor: Neighbor) = async {
            if x.GetCoordinator x.Round = neighbor then
                let nack: ConsensusNegativeAcknowledgement = {round = x.Round}
                do! x.server.SendMessage nack neighbor.host neighbor.port
                do! x.StartConsensus ()
        }

        // ...
```

Take a look at Consensus.fs for more details.

Run And Experiment

To run the distributed systems, learn about how failure detectors and consensus work, open Program.fs and experiment with the settings.

```
// ...

let messageHandling message = async {
    printfn "User message handling here..."
}

let startConsensus (node: Node) = async {
    // Start consensus after 10 seconds
    do! Async.Sleep 10000
    do! node.StartConsensus ()
}

let getHostAndPort (address: string) =
    let hostport = address.Split([|':'|])
    { host = hostport.[0]; port = int hostport.[1] }

[<EntryPoint>]
let main argv =
    printfn "Hello from the F# Distributed System!"

    let printUsage () =
        printfn "USAGE:"
        printfn "1st arg is host and port to run this node on."
        printfn "2nd arg is comma separated list of hosts and ports of neighbors."
        printfn "3rd arg is a default value to start the node with."

    if argv.Length = 0 then
        printUsage ()
```

```
else
    if argv.Length < 1 || argv.Length > 3 then
        printUsage()
    else
        let nodeInput = Array.get argv 0
        let node = getHostAndPort nodeInput

        let startingValueConf, neighborsConf =
            match argv.Length with
            | 1 -> ("Default", [])
            | 2 ->
                let neighborsInput = Array.get argv 1
                let neighbors = neighborsInput.Split([|','|])

                                |> Seq.map getHostAndPort |> Seq.toList
                ("Default", neighbors)
            | 3 ->
                let neighborsInput = Array.get argv 1
                let neighbors = neighborsInput.Split([|','|])

                                |> Seq.map getHostAndPort |> Seq.toList
                (Array.get argv 2, neighbors)
            | _ -> failwith "This shouldn't ever happen."

        printfn "Starting Node %s:%i..." node.host node.port

        let nodeInstance = Node.Node(node.host, node.port, startingValueConf)
        let nodeConfiguration = {
            networkProtocol = UDP // or TCP
            failureDetector = HeartbeatSuspectLevel // or PingAck,
                                                    // SimpleHeartbeat,
                                                    // HeartbeatRecovery,
                                                    // HeartbeatSlidingWindow,
                                                    // HeartbeatSuspectLevel
            consensus = ChandraToueg
            receiveMessageFunction = messageHandling
            neighbors = neighborsConf
            gossipping = true // or false
            verbose = false  // or true
        }

        nodeInstance.InitializeNode(nodeConfiguration) |> Async.RunSynchronously
        nodeInstance.Start |> Async.RunSynchronously

        // Uncomment to experiment with consensus
        // nodeInstance |> startConsensus |> Async.RunSynchronously

        Thread.Sleep 600000
        0
```

Keep in mind, this is not a bullet-proof distributed system, do not use it in prodiction. It is for illustrative and educational purposes only.

To run the example, make sure you specified the configuration setting you prefer, open several terminals, and start the project in each of them:

```
dotnet run \
    "nodehost:nodeport" \
    "neighbor1host:neighbor1port,neighbor2host:neighbor2port,neighborNhost:neighborN:port" \
    "value"
```

Make sure to replace the host, port, and value with your own information.

To stop any of the projects press Ctrl+C in the terminal.

To build the solution, go to the src directory and execute build-fsharp.sh (for Linux and Mac).

I hope you enjoyed the series!