Assignment 1 – v2

Assignment 1 requires **one** practical **project**: An implementation of the **Echo distributed broadcast** algorithm, extended to determine the **size of the network**.

The network is described by a text configuration file. For example, the sample network used in the lectures can be described by the following text, say **config4.txt**:

```
// string -> int dictionary
// nodes
1 8081 // node 1 port 8081
2 8082
3 8083
4 8084

// arcs
0 8090  // arc port 8090
- 10  // default delay
1-3 3000
1-4 6000  // arc 1-4 delay 6000ms
2-1 1000
3-2 1000
4-3 1000
```

In fact, this config file describes a conceptual string-to-int dictionary, to be used by all instances of your solution, with the following interpretations:

- Node 1 is a process listening at port 8081
- …
- Arc "node" 0 is a process listening at port 8080 (more about this later).
- For unspecified arcs, the default static delay is 10ms (immutable).
- Arc 1-3 has a static delay of 3000ms (immutable).
- …
- ~~The arcs that are not explicitly listed have a default delay of 1ms (immutable).~~

The config file may contain comments and extra spaces or lines. Comment, blank and empty lines must be discarded, as well as extra spaces. This could be the actual dictionary parsed from the above sample config file.

Dictionary<String,Int32> (11 items)

Key	Value
1	8081
2	8082
3	8083
4	8084
0	8090
-	10
1-3	1000
1-4	1000
2-1	1000
3-2	1000
4-3	1000
	45430

The node and arc process should be called **node.exe** and **arc.exe**, respectively. These will be started in the following way, to ensure that stdout's are recorded:

```
rem 2^>^&1
start "node1" cmd /k node.exe %config% 1 2 3 4 ^> node1.log
start "node2" cmd /k node.exe %config% 2 1 3 ^> node2.log
start "node3" cmd /k node.exe %config% 3 1 2 4 ^> node3.log
start "node4" cmd /k node.exe %config% 4 1 3 ^> node4.log

rem 2>&1
arc.exe %config% > arc.log
```

To help the markers, stderr will still be sent to the command console, but we will duplicate the same messages to stdout.

The first command-line argument of all processes is the config file. Nodes have one or more numerical arguments: the first is the node number, followed by adjacent node numbers.

We recommend that messages have the following high-level "wire" format (WCF):

```
[ServiceContract()]
public interface INodeService {
   [OperationContract(IsOneWay=true)]
   [WebGet()]
   void Message(int from, int to, int tok, int pay);
}
```

Nodes (including the arc "node") should print incoming and outgoing messages using the following formats, **exactly** (but allowed extra spaces and additions at the end**)**:

```
Console.WriteLine($"... {Millis():F2} {ThisNode} < {from} {to} {tok} {pay}");
Console.WriteLine($"... {Millis():F2} {ThisNode} > {from} {to} {tok} {pay}");
```

Where Millis() is function which returns time of the day in milliseconds (or higher precision):

```
Func<double> Millis = () => DateTime.Now.TimeOfDay.TotalMilliseconds;
```

You can write out other lines, if you wish/need so, but please do **NOT** write any other lines with the same four chars prefix "… " (three dots and one space). This convention will be used to trace the overall message flow, and determine its consistency.

In the above WriteLine calls:

- ThisNode : the current node number, receiver or sender
- from : the sender node number
- to : the target node number
- tok : token kind, **1** for forward, **2** for return (we could use a single token in both directions, but having distinct values adds in clarity)
- pay: the payload, here the computed size (partial or total)

In our sample scenario, node numbers are in the range 0..4, where 0 is the arc "node".

Let's now discuss the critical roles of the **arc.exe** process. We can also call it "network manager" or "asynchroniser". This process will intercept all messages and forward them after the associated delays, as defined in the config file.

In other words, the conceptual message from node x to node y will follow a longer path: node x -> arc -> node y, with a required time delay being added in the arc process.

Arc process 0 is started **last** and will start the echo algorithm, by sending a first forward message to the already started node process 1. The distributed algorithm will end after node process 1 sends a return message to arc process 0 – the payload of this last message should contain the size of the whole network.

Thus, arc process 0 plays a critical double role:

- Intercepting all node messages and ensuring their specified delays
- Acting as dummy parent for the root node 1

For consistency (and fair marking), we require that these process (node and arc) are implemented in C# or F#, using WCF (Windows Communication Foundation). Each process must be completely developed in one single file. The files must be respectively called **node.cs** and **arc.cs** (C#), or **node.fs** and **arc.fs** (F#). Please only use standard libraries extant in the labs (no other libraries are allowed). However you develop these, the files must be compilable using the command lines compilers, e.g.

```
csc /r:System.ServiceModel.dll /r:System.ServiceModel.Web.dll node.cs
csc /r:System.ServiceModel.dll /r:System.ServiceModel.Web.dll arc.cs
```

Please submit your two source files to the university **ADB assignment dropbox**, following the instructions on **Canvas** and **ADB** itself. Please ensure that your submission is compilable and runs properly in our labs (the same environment which will be used by the markers).

Appendices

A1-WCFdemo-HelloClientServer: basic client server

A1-Ping-Pong-demo: client+servers (P2P)

Uses an explicit **OperationContextScope** to avoid a possible dialogue bug

A1-Echo-Model-Labs: lab results of the model solution