SIMULATION OF THE FORD–FULKERSON ALGORITHM USING OMNET++

Paula SEVASTIAN Andrei Bogdan RUS Virgil DOBROTA
Technical University of Cluj-Napoca, Communications Department, 400027 Cluj-Napoca, Romania,
Phone: +40-264-401226, Fax: +40-264-597083,
Emails: paula_sevastian@yahoo.com, {Bogdan.Rus,Virgil.Dobrota}@com.utcluj.ro

Abstract: This paper presents a contribution to the OMNeT++ 4.1 simulator in C++ by integrating our own implementation of the FF - Ford-Fulkerson multipath routing algorithm. In order to validate it, an eight-node testbed works with dynamic requested and released flows on each link, updated every second. Four simple scenarios were proposed, considering all possible combinations of fixed and/or random variation laws. Additionally, the path search algorithm BFS - Breadth First Search was implemented too. This software package running under Fedora Core gets the input data from three files called Nodes, Connections and Params. This mechanism provides the interworking between the C++-based FF simulator and the Java-based application in Intellij IDEA 10.5, used as traffic variation simulator. The graphical representations of the occupied, the requested and, respectively, the released flows are obtained by aggregating the contributions of all flows from the paths for any given source–destination pair.

Keywords: *Breadth First Search, Ford - Fulkerson, multipath routing algorithm, OMNeT++*

I. INTRODUCTION

The maximum flow determination has been a long term issue in several areas (including networking) for many years. Roughly speaking, it calculates the maximum amount of "stuff" that can be moved from one part of a network to another, being aware about the capacity limitations of the links. This "stuff" could be data packets travelling over the Internet, the water travelling through pipes or some trucks travelling on the highways. Thus the links limitations could refer to the bandwidth (for the Internet connections), the pipe dimensions (for the water distribution systems) and the average traffic speeds (for congested roads). The graph theory is offering a mathematical support to solve the problem of the max flow. One of the nodes (we prefer these terms instead of vertices) in the graph is called the source node, whilst another one is the destination (or sink). Each link (or edge) in this graph has an associated capacity. Limiting now the area of discussion to the current Internet, two arguments are driving the investigations proposed in this paper: a) the increased demand for higher transfer rates; b) the computational time for optimal routing decisions. A good candidate to help the well-known single path mechanisms in place nowadays is the multi-path packet forwarding.

Similar studies were carried out over the years. For instance an extended version called RMF (Randomized Max Flow) was investigated for EH-WSN (Energy Harvesting Wireless Sensor Networks). This network uses the nodes which are able to harvest power from the environment giving them theoretically unlimited power for a maximal exploitation. The paper [6] introduces the problem of energetic sustainability and the concept of maximum energetically sustainable workload used in order to optimize the routing algorithms for EH-WSNs. Within the RMF the routing tables are calculated off-line in order to reproduce the optimal flow distribution provided by the max-flow approach. This is why the optimal flow values were directly annotated in the routing tables associated with each node. Similar studies could be found in [7], [8]. A max flow multipath scheme based on Ford-Fulkerson, presented in [2], was designed to reduce latency, to provide high throughput and to balance the traffic load. It determined a set of disjoint paths that are loop free with maximum flow, then splitting network traffic among those paths. The simulation proved that this solution performed better than a multi shortest path scheme.

Our approach refers to the behavior of a simulated network using FF (Ford-Fulkerson) routing algorithm while a traffic variation simulator constantly modifies the existing flows. The algorithm computes the maximum available flow for a given source-destination pair. Note that the result is not influenced by the paths that are used to compute the max flow. The paper presents an original implementation of Ford-Fulkerson in OMNET++, which relies on real-time quality of service information obtained for each node using cross-layer techniques. However the interaction with this module is not covered herein, but it is related to our previous work published in [9]. The rest of the paper is organized as follows: Section II covers the theoretical aspects of the Ford-Fulkerson and BFS (Breadth First Search) algorithms. The design principles of their OMNET++ implementations are discussed in Section III, followed by experimental results in Section IV. Conclusions, further work and the references are ending the paper.

Manuscript received January 31, 2012; revised May 9, 2012

II. THE FORD-FULKERSON AND THE BREADTH FIRST SEARCH ALGORITHMS

Let us define two basic concepts used by any max-flow algorithm: residual path and augmenting path. *A residual network* has exactly the same nodes as the original network, and one or two links for each original link. If the flow along the link *ij* meets the first two flow restrictions, there is a forward link *i j* with the capacity equal to the difference between the capacity and the flow and a backward link *ji* with the capacity equal to the flow on *ij*. These maximum flows are called residual capacities. *An augmenting path* is a path from the source to the sink in the residual network, having the purpose to increase the flow in the original network. The path capacity of the augmenting path is the minimum capacity of a link along it and represents the value with which the flow in the original network will be increased. The Ford-Fulkerson algorithm is briefly defined as follows: it starts with no flow everywhere in the network and it increases the total flow as long as there is an augmenting path from source to destination in the residual network, with available capacity on all links. In other words, as long as there is a path from a source to a destination, with available capacity on all links, the flow can be sent along that path and so on. The paths with available capacity are the augmenting paths.

Let us have the following notations: *G* is the graph for the original network, *S* the source node, and *D* the sink/ destination node. For the graph G_f corresponding to the residual network C_f (p) is the capacity for the augmenting path, i.e. the minimum of the link capacities c_f(u,v) for all links belonging to path p. Note that *C* and *F* represents the link capacity, respectively the maximum flow from *S* to *D*.

```
1.  f(u,v)=0, ∀ link(u,v) in the original
    network, where f(u,v) is the flow
2.  while ∃ path from S to D in Gf such that
    cf(u,v)>0, ∀(u,v) ∈p
       i)    Cf(p) = min {cf(u,v)|(u,v) ∈ p}
       ii)   f(u,v)->f(u,v)+Cf(p)
       iii)  F->F+Cf(p)
    where    Cf(u,v)=c(u,v)-f(u,v)
```

When no more paths are found in step 2, it means that there are no more augmenting paths from S to D in the residual network. This means that the maximum flow has been found. The FF algorithm has the advantage of getting the correct result (i.e. the maximum flow) no matter how the sub-problem of finding the augmenting paths is solved. We implemented FF according to the following pseudo-code:

```
int compute_max_flow()
    max_flow = 0
    while(true)
        bfs (start_node)
        if (augmenting path was found using BFS)
            path_capacity =
            get_augmenting_path_capacity()
            max_flow += path_capacity
        else
            exit while
    end while
    return max_flow
end
```

The most commonly used algorithms for traversing a graph and searching a node in a graph are: BFS (Breadth First Search) and DFS (Depth First Search). The DFS traverses the graph in such a way that it tries to go as far as possible from the root node. In this paper we chose the BFS algorithm, because when DFS is combined with FF, the performances are very poor [1], [2].

The BFS aim is to traverse the graph as close as possible to the root node, a queue being used for implementation. BFS visits the nodes level by level, starting from the root level (level 0), as in Figure 1.

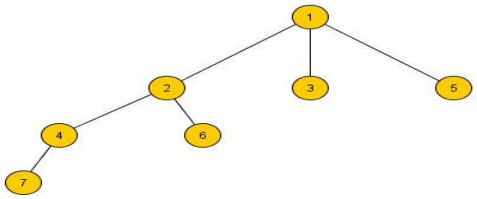

Figure 1. Graph to illustrate the BFS algorithm

If the node 1 the root, the order in which BFS traverses the graph is 1-2-3-5-4-6-7. Whenever a node from the graph is visited, all its neighbors are added in a queue and the node is marked as visited. Then the next node from the queue is popped and the process is repeated. Thus the traversing of the graph on levels is assured. Extra care should be given to the marking of a node as visited when the node was firstly visited in BFS. If this rule is not accomplished, the property that BFS finds the shortest path from source to destination could be lost. The BFS implementation followed the pseudo-code:

```
void bfs (start_node)
    queue q;
    push start_node in q
    while(q is not empty)
        top= first element from the queue
        for each unmarked neighbor node w of top
            mark w
            add w to end of q
            if w is destination node
              terminate
            else
              continue to next iteration
        delete first element from the queue
    end while
end
```

III. THE OMNET++ IMPLEMENTATION

Our implementation of Ford Fulkerson algorithm includes two main applications: an OMNET++-based testbed and a Java-based traffic variation simulator.

The first part was written in C++ and realizes a network builder and a network configurator. This means that we chose to build the network dynamically. Thus the information needed is read from two input files: *Nodes* and *Connections*. The *Nodes* file contains information related to the nodes (routers or hosts in general, just routers in our case). On each line the following parameters are provided: node index, node name, router IP, type of node (router type/ OMNeT++ module type) and the position of the node in the

network given by the coordinates x and y:

```
1 S 192.168.1.1 project.node.FFRouterGen 230 30
```

The node with index 1 has the name *S,* the IP address 192.168.1.1 and it is an *FFRouterGen* module type. Its position in the network is given by the point of coordinates (*230, 30*). The *Connections* file contains information related to the way the routers/nodes are connected to each other. Each line in the file contains the indexes of the nodes that are connected, the names of the interfaces on which they are connected, as well as the IP addresses of those interfaces. All the connections mentioned in this file are unidirectional:

```
1     ppp0     192.168.2.1
2     ppp0     192.168.2.2
```

The interface *ppp0* of node with index 1 and IP address 192.168.2.1 is connected to interface *ppp0* of node with index 2 and IP address 192.168.2.2. For this application, *Connections* file contains unidirectional links between the nodes defined in *Nodes* file. The network on which FF algorithm is simulated has exactly 8 nodes: a source router, a destination router and 6 routers.

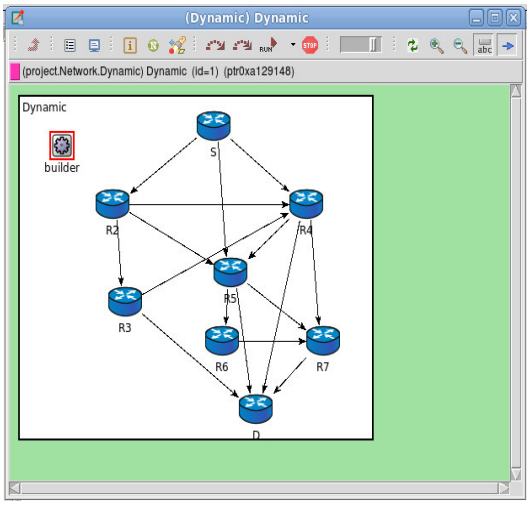

Figure 2. Dynamic network built from files.

For the implementation we have created two types of modules files: a generator router called *FFRouterGen.ned* and a simple router module called *FFRouter.ned.* Both modules use two double variables that represent the *x* and the *y* coordinates of the node. Their values are not set within the configuration file, since they are read from file and set whenever each node is created. These types of modules have vector gates of *inout* type. After the network was built, the role of the network configurator comes in. This is the part where the FF algorithm is applied, and it uses *Params* as input file containing the available capacity and the occupied flow for each link.

Within the first iteration, the values of the flows are all zero. During the simulation these flows are modified. This part of the application also generates an output file called *Results* including all the augmenting paths and their capacities found by BFS.

The *Params* file contains information on the links from the network. We presume the available capacity and the occupied flow are specified for each link in the network. An example of line from this file is the following:

```
1        2              10000                  0.0
```

The links are given by the indexes of the nodes (specified in *Nodes* file). In this example, the link between node with index 1 and node with index 2 will have the maximum capacity 10000 kbps and the initial occupied flow 0. If we check *Nodes* file, we will see that this is the link between S (source) and router R2. The *Results* file contains the results given by the FF algorithm simulation with BFS for finding the augmenting paths. A line from the file has the following form:

```
S-Router_i-Router_j-…-Router_k-D Path_Capacity
```

If the algorithm does not find any augmenting path, then the file will be empty.

The traffic variation simulator is a Java application in Intellij IDEA 10.5 that uses *Nodes*, *Params* and *Results* as input files. It computes a new occupied flow for all the links in the OMNeT++ network. Then it modifies the *Params* file by overwriting the old flow values with the new ones. *Params* is the input file for the OMNeT++ simulation of FF algorithm. By choosing either button *Fixed* or *Random,* the user may insert a fixed or a random value (in percentages) to be used in order to compute the new flow. There are two types of values: *Requested Flow* and *Released Flow*. In order to compute the new value of the flow, the following formula is applied:

$$F_{ij\_currently\_occupied} = F_{ij\_previously\_occupied} + Requested\ Flow - Released\ Flow \quad (1)$$

where: $F_{ij\_previously\_occupied}$ represents the value of the flow from node *i* to node *j* read from the input file *Params;* $F_{ij\_currently\_occupied}$ is the current flow that will occupy the *i- j* link in the next OMNeT++ simulation of the FF. This is computed by adding some value that has the significance of a *Requested* flow and subtracting another value with the significance of *Released* flow. *Requested Flow* refers to a percentage of the maximum flow for each link computed with FF and BFS. It actually represents how much flow from the maximum occupied is going to be added to the current (previously) occupied flow for the next simulation of FF algorithm. Supposing *x* is the value inserted by the user, then the *Requested* flow can be expressed as:

$$Requested\ Flow = x\% * F_{ij\_augmenting} \quad (2)$$

$F_{ij\_augmenting}$ represents the value of the flow that is read from the *Results* file. Remember that every line contains the augmenting path and its corresponding flow. *Released Flow* refers to a percentage from the previously occupied flow on each link. It represents the quantity of flow that can be subtracted from the current (previously) occupied one.

Suppose *y* is the value inserted by the user, then the formula is the follwing:

$$Released\ Flow = y\% * F_{ij\_previously\_occupied} \quad (3)$$

If we choose the *Fixed* button, the values used to compute the new occupied flow are that ones inserted by the user. If the *Random* button is pressed, then the values used by the application to compute the new flow are random values between 0 and those inserted by the user.

IV. EXPERIMENTAL RESULTS

The purpose of the OMNeT++ simulation application and of the traffic variation simulator is to study the way flows behave and modify in a given network. We are referring to the simulation of the FF algorithm that constantly maximizes the flow that can be sent across the network. The traffic variation simulator constantly modifies the occupied flow across the links of the network.

The Java GUI application has the additional role to write data into a .csv extension file called *ParamsHistory*. Data written in this file is represented by all the new flow values and the requested flow ones. By using this file we actually keep track on the changes in the flows.

The OMNeT++ application computes the un-occupied flow that can be sent across this network every one second, using the flows from the input file *Params*. The Java application modifies the flows it every one second, in this way simulating a traffic variation. These represent the occupied flows in each link/connection from the given network.

The initial conditions for the network are the following: all initial flows are 0 and all links have a capacity of 10 000 [kbps]. Within the first iteration of the simulation there will be no flows everywhere, since we want to find out the maximum flow that can be sent from the source across the network. For the next iterations, the occupied flow will not be zero anymore, but we still want to see how much can the path be augmented.

Scenario 1 - Fixed Requested Flow & Fixed Released Flow: This is not very realistic but it has been used to calibrate the OMNET++-based implementation. Every second, the new occupied flow is computed using the same percentages from the available flow and from the previous occupied one. This formula used to compute the new or current occupied flow actually implements the behavior of a negative feedback (that ensures the stability of the system). After a short period of time (a few seconds) the value of the released flow becomes equal to the value of the requested flow, the current occupied flow from the links becoming constant.

Scenario 2 - Fixed Requested Flow & Random Released Flow: The Java application computes the new occupied flow using the same percentage for the requested flow and a random percentage for the released flow. In this scenario, the requested flow will never get equal to the released one. Figure 3 corresponds to flow values computed for a fixed value of 10% from the available flow and a random value from 0 to 80% from the previously occupied flow. Observe that most of the time the released flow is smaller than the requested one. This is a case when the links will not be fully occupied and there will be always available flow to be used to augment the occupied flow.

Taking into consideration the quantity of the released flow is not the same for the consecutive iterations. Thus the variation of the occupied flow depends highly on the variation of the released flow. The requested flow does not vary that much as the requested one. Anyway, for high values of released flow, the occupied one decreases whereas for small values, it increases visibly.

Taking into consideration the quantity of the released flow is not the same for the consecutive iterations, it can be seen that the variation of the occupied flow depends highly on the variation of the released flow. The requested flow does not vary that much as the requested one. Anyway, for high values of released flow, the occupied one decreases whereas for small values, it increases visibly.

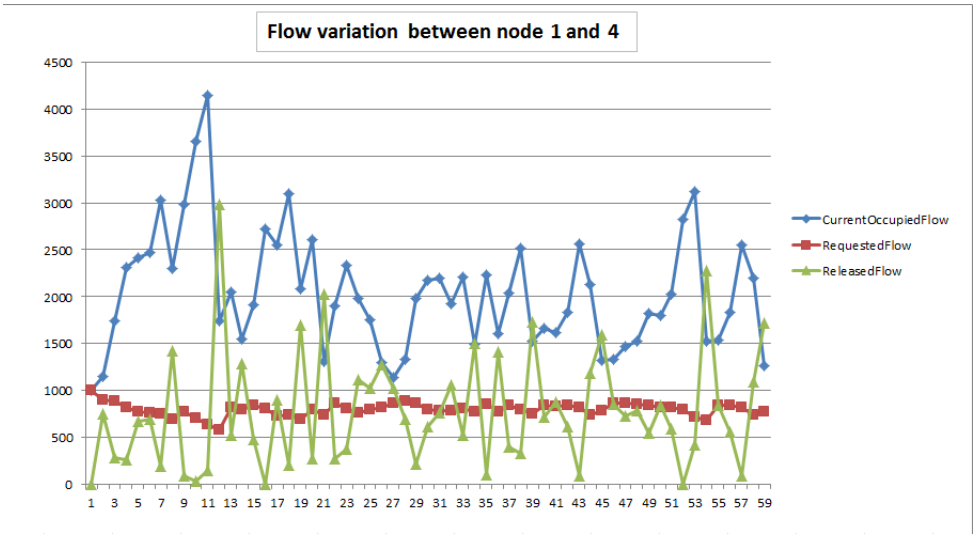

Figure 3. Scenario 2: current occupied flow, released and requested flows

There is a case where *no more augmenting paths* can appear but, due to the randomness of the released flow, the values of the flows will modify immediately. In order to graphically demonstrate this situation, we chose a high value as the percentage from the available flow (95%), which is a fixed value, and a small value as the maximum percentage from the previously occupied flow (2%). The variations of the occupied, requested and released flows is presented in Figure 4. The value -1 for the requested flow means that the link was fully occupied and the FF algorithm could not find any more augmenting paths. The value of the requetsed flow is -1 within [26,30s] interval, meaning that the link is fully occupied and there is no more flow to augment. This also happens because the released flow is 0 in that interval. When flow is released, the value of the requested flow immediately becomes positive. This happens automatically, since the released flow represents a very small random percentage from the previously occupied flow.

Scenario 3 - Random Requested Flow & Fixed Released Flow: The percentage from the available flow needed to compute the requested flow is a random number and the value of the percentage used to compute the released flow is a fixed number. For this scenario, the only way to obtain a situation were no augmenting paths are found (the occupied flow is maximum) is the situation in which the released flow is 0.

Figure 5 corresponds to a large domain for percentages to compute the requested flow (from 0% to 40%) and to a high value to compute the requested flow (75%). In this scenario, there will always be some released flow after each simulation, so FF with BFS algorithm will always find paths on which the flow can be augmented. The only way to produce a maximum occupied flow or to fully use the resources of the network within this scenario is to reset the value for the released flow. The variations of the flows can be seen in Figure 6.

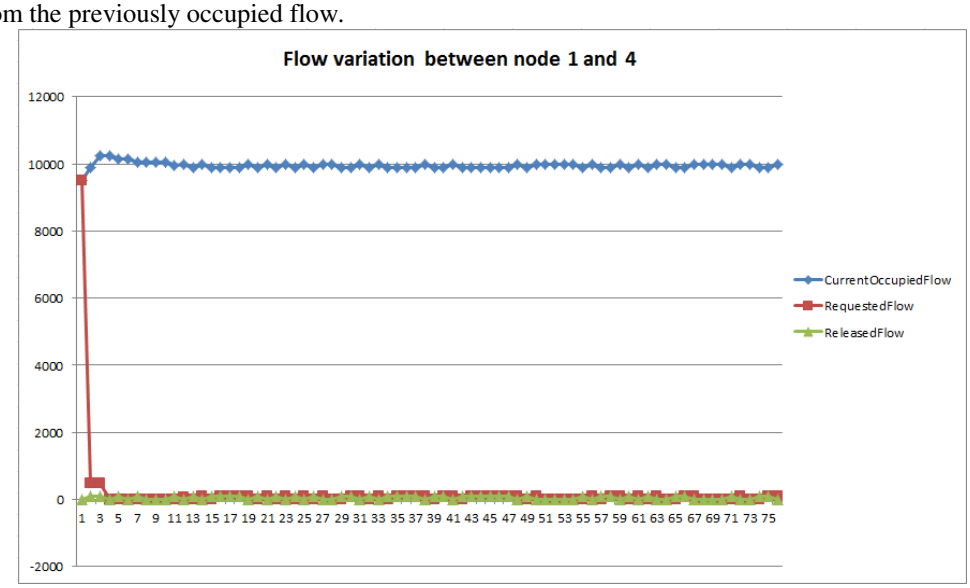

Figure 4. Scenario 2: high percentage for requested flow and low percentage for released flow.

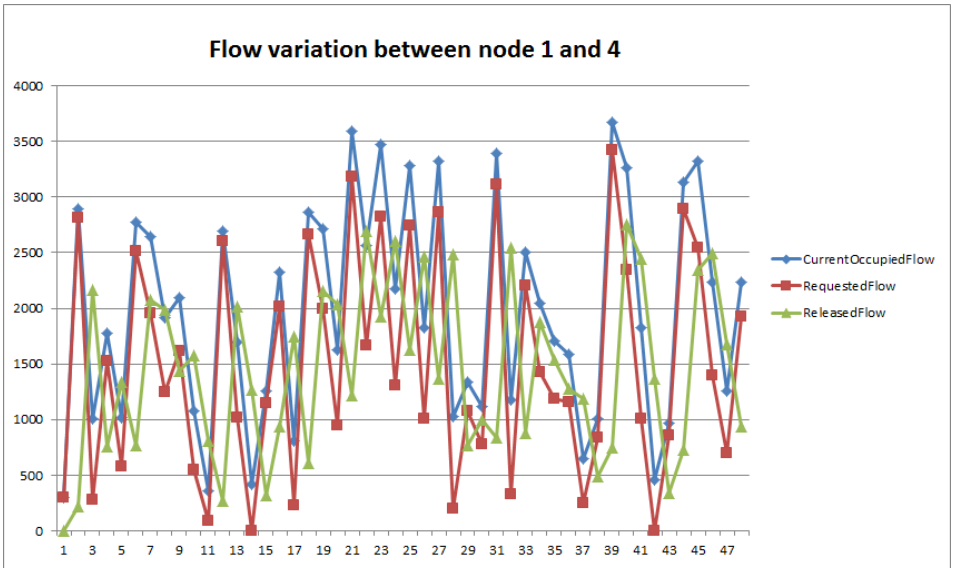

Figure 5. Scenario 3: the occupied, the requested and the released flows.

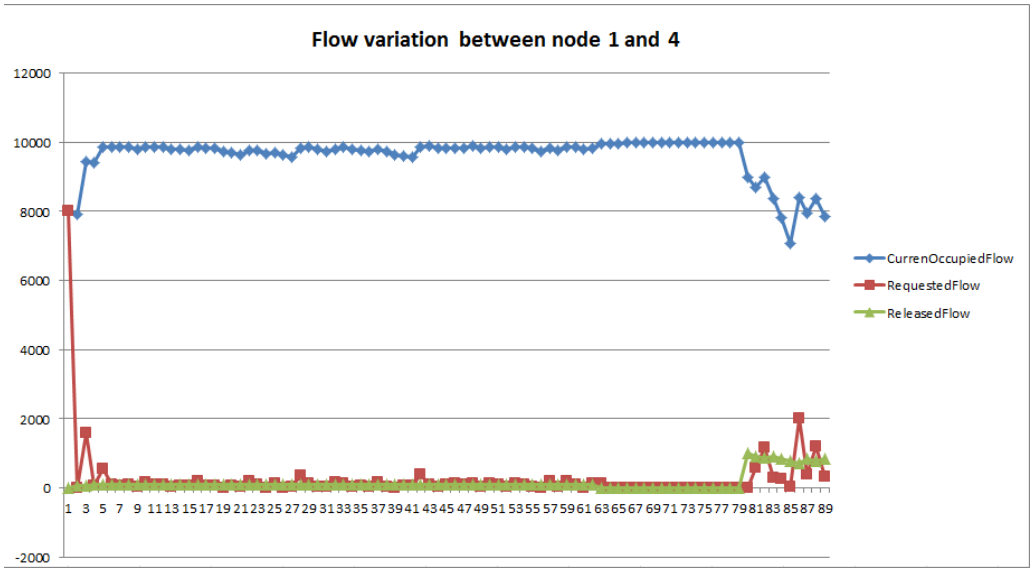

Figure 6. Scenario 3: the occupied, the requested & the released flows (initially zero, later on non-zero).

To show the case when if the value of the occupied flow is maximum, no augmenting paths are found, in Figure 7 we plotted the variation of the requested flow for interval [1,16s].

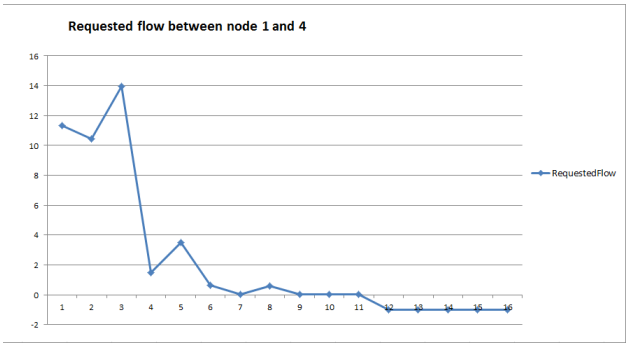

Figure 7. Scenario 3: the requested flow for [1,16s].

Because the released flow is 0 for that interval, the value of the available flow decreases (with small variations) until it gets to 0. This is when the requested flow is equal to -1 and it means that there are no more paths found. The interval [18,27s] in Figure 6 corresponds to a non-zero percentage value used to compute the released flow and values belonging to [0, 65%] interval for the requested flow. It can be seen that the occupied flow decreased immediately and available flow appears again in the network

Scenario 4 - Random Requested Flow & Random Released Flow: Both the percentage value used to compute the requested and released flows are random values. This is the case when it is quite hard to find moments when there are no augmenting paths. Figure 8 shows the random variation of the occupied flow, whilst Figure 9 presents the variation of both the requested flow and of the released one.

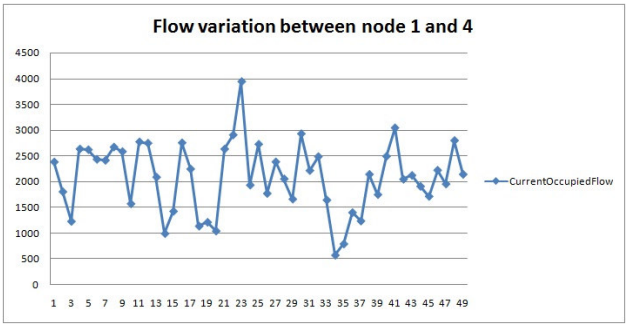

Figure 8. Scenario 4: the occupied flow.

In this scenario it is the least probable to obtain a maximum traffic flow for the whole network. Due to the random percentages, there will always be a released flow and the occupied one will be different for every simulation. The worst case that could happen would be the situation in which the value of the requested flow is very high and the one for the released flow. This case could lead to a flow in the network that could not be augmented in the next iteration. But this situation will be remediated within the next second of time, when different percentages from the occupied and from the available flows are computed.

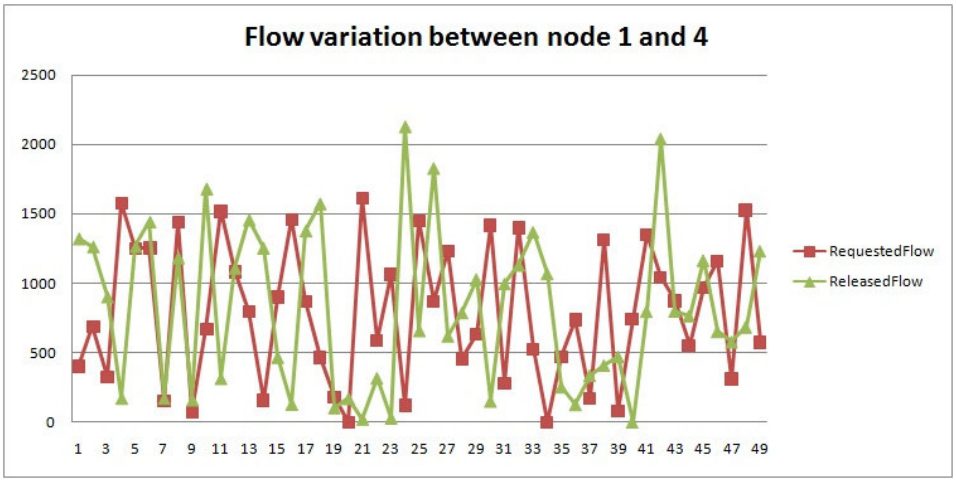

Figure 9. Scenario 4: the requested & the released flows

V. CONCLUSIONS

The multipath routing processes for a given network involving Ford-Fulkerson algorithm demonstrates its benefits in computing the maximum available link flows and, implicitly, the total available network flow. The solution took advantage of the BFS algorithm to find the shortest paths from a source to a given destination. The flow of each path found using BFS can be augmented with the minimum available flow of all the links belonging to that path. The process of finding paths with BFS is repeated until no more augmenting path can be found. When this happens, we can state that Ford-Fulkerson algorithm found the maximum available flow of that network. The experiments covered all four possible combinations of fixed and/or random variation laws for the requested and released flows.

Although we can give an interpretation of the results, our main purpose was to validate the software packages and the formulas proposed herein. Our future work envisages the involvement of more realistic approaches: a) the variation laws should be according to real traffic models (e.g. Pareto); b) the values for the link flows should be eventually obtained by cross-layer techniques from a real network. In this latter case, the FF implementation could be detached from OMNET ++ simulator and partially the code could be reused in real routers.

REFERENCES

[1] H.T. Cormen, E. C. Leiserson, R.L. Ronald, S. Clifford, *Introduction to Algorithms*, Second Edition, MIT Press, 2001

[2] R. M. Ahmed, J. Fratwell Rod, C. Brahem, "MFMP: Max Flow Multipath Algorithm", *IEEE Second UKSIM European Symposium on Computer Modeling and Simulation*, 8 Sept.2008

[3] H. D`Arcy, "Intellij IDEA", *PC Magazine*, No.52, 2009

[4] A.B. Rus, *Quality of Service Through Cross-Layer Techniques for the Future Internet*, Technical University of Cluj-Napoca, 2011, http://www.etti.utcluj.ro/download/678_Rezumat_te.pdf

[5] A.Varga, *OMNeT++ User Guide,* http://www.omnetpp.org/documentation, 2012

[6] E. Lattanzi, E. Regini, A. Acquaviva, A.Bogliolo, "Energetic sustainability of routing algorithms for energy-harvesting wireless sensor networks". *Computer Communications*, Vol.30, Issues 14-15, 15 October 2007, pp 2976-2986

[7] D. Hasenfratz, "Simulative Analysis of Routing Algorithms for Energy Harvesting Sensor Networks". *Swiss Federal Institute of Technology*, Zürich, 2009

[8] D. Hasenfratz, A. Meier, C. Moser, J. J. Chen, L. Thiele, "Analysis, Comparison and Optimization of Routing Protocols for Energy Harvesting Wireless Sensor Network". *SUTC'10: Proc. of the 3rd Conf. on Sensor Networks, Ubiquitous and Trustworthy Computing*, 2010, pp 19-26.

[9] A.B. Rus, V. Dobrota, A. Vedinas, G. Boanea, M. Barabas, "Modified Dijkstra's Algorithm with Cross-Layer QoS", *Acta Technica Napocensis, Electronics and Telecommunications*, ISSN 1221-6542, Vol.51, No.3, 2010, pp. 75-80.