Optimality of the Tree Building Control Protocol

Ondrej Dolejs
CAK-DCE, Faculty of Electrical Engineering
Czech Technical University
Karlovo nam. 13, Prague, Czech Republic
xdolejs@lab.felk.cvut.cz

Zdenek Hanzalek,
CAK-DCE, Faculty of Electrical Engineering
Czech Technical University
Karlovo nam. 13, Prague, Czech Republic
hanzalek@rtime.felk.cvut.cz

Abstract

This text focuses on the simulation and evaluation of Tree Building Control Protocol (TBCP) described in [1]. The TBCP is distributed algorithm used for building a spanning tree over TBCP entities.
In this paper, the main part of the TBCP algorithm called join procedure is explained and the state machine of the join procedure is presented. Then, the non-distributed Branch and Bound algorithm finding an optimal spanning tree is shown. Finally, the TBCP is simulated in OPNET Modeler and compared to the optimal spanning tree.

1 Introduction

When it is needed to send the same data from the *source* node to the destination group, multicast is more efficient than unicast where separate copy is sent to each destination. In multicast the data travel from the *source* through the network and in the fork node the copies of the data is send toward destination nodes.

The multicast reduces the transmission overhead on the sender side and it can reduce network load, therefore the destination nodes can receive the data earlier (depending on implementation). The multicast is identical to a broadcasting, if the destination group includes all the network nodes.

The tree generated by the TBCP algorithm is used for multicast application that belongs to the Application Level Multicast. Therefore, no prior knowledge of the network topology and parameters is needed for the construction of the TBCP tree. The algorithm is based on its own measurements of the Round Trip Time (RTT). Optimality criterion in this article is based on the delay measured from the *source* node to the last node in the destination group.

2 Related work

The multicast routing tree algorithms can be divided into two main parts. If the optimality criterion is a cost of the constructed tree, this problem is called minimum spanning tree (MST). There are many well known algorithms to find MST e.g. Kruskal's or Prim's algorithm [6].

The shortest path tree (SPT) algorithms are used, if the optimality criterion is delay on the other hand. The delay is the sum of link delays on the path (from the *source* node to any other node). The graph algorithms (e.g. Dikjstra [3], Bellman-Ford [8]) can be used for solving the SPT problem.

Another classification of the multicast routing tree algorithms is the place, where the algorithm is executed (centralized or distributed algorithms), and the sensibility to the network changes (e.g. link fails). The centralized dynamic versions of the Bellman-Ford, D'Esopo-Pape and Dijkstra algorithm are presented in [7]. The main disadvantage of centralized algorithms is the problem with central node failure, heavy load of the central node and a need of network topology knowledge in the central node.

The TBCP is a distributed algorithm for constructing multicast routing tree. It was designed in GCAP project at Lancaster University [1].

3 TBCP protocol

A crucial part of the TBCP protocol is the JOIN procedure: when the *newcomer (N)* wants to join the TBCP tree, it sends a *HELLO* message to the *source* node. The *source* sends the list of its existing *children(C_i)* in a *HELLO_ACK* message back to the *newcomer*. When the *newcomer* receives a *HELLO_ACK* message, it makes a distance measurement (RTT) from itself to all *source's children* and between itself and the *source* node. The table of the RTT measurement is sent to the *source* in a *JOIN* message by the *newcomer*. If the *source* receives a *JOIN* message from a *newcomer (N)*, then the *source* first calculates a score function (SF) based on RTTs and then the *source* decides, where the *newcomer* will be placed. Three configurations are possible. In the first case (SF=1) *N* goes directly under the *source*, in the second case (SF=2) *N* goes directly under the *i-th child (C_i)* and in the third case (SF=3) *N* goes directly under the *source* and the *i-th child (C_i)* is redirected under another *child* or *newcomer*. In the second and third case it is needed to send a *HALLO* message to a new *candidate parent (P)*.

The whole JOIN procedure is repeated in the same way as explained above (the *candidate parent* acts as the *source* in the previous text and the *newcomer* is a *newcomer* or a redirected *child*). The illustration of JOIN procedure is in Fig. 1.

The TBCP is a distributed algorithm. The main disadvantage is that the *parent* node has the RTTs only from one layer (between itself and a *newcomer*, between its *children* and a *newcomer* and between itself and its *children*). It is clear that this algorithm is not optimal as the *parent* has no RTTs from the next lower layers.

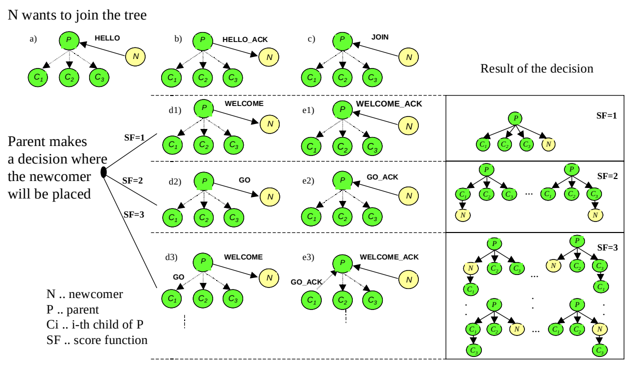


Fig. 1 Join procedure of TBCP protocol

3.1 Implementation of the TBCP protocol in OPNET Modeler

OPNET Modeler is based on a series of hierarchically related editors e.g. Network, Node and Process editor that directly model the structure of actual networks. For more details on the network modelling and optimisation, please refer to [4]. The TBCP state machine implementation in OPNET Modeler 8.0.C. is shown in Fig. 2.

The TBCP state machine (Fig. 2) could be divided into three main parts:

1/ *The newcomer state machine:* it processes the new request to join the TBCP tree and consists of four states*:*

- N_HELLO– either after events *REQUEST_FROM_API* or after sending a *GO_ACK* message to the previous *candidate parent*, the *newcomer* sends a *HELLO* message to the *candidate parent*
- N_HELLO_ACK – after receiving a HELLO_ACK message with a list of the *candidate parent's children*, the *newcomer* makes a distance measurements (RTT) and sends the results in a JOIN message to the *candidate parent*
- N_GO – after receiving a GO message, which contains a new *candidate parent*, the *newcomer* sends a GO_ACK message back to the last *candidate parent*

N_WELC_ACK – after receiving a WELCOME message the *newcomer* saves the *candidate parent* as its *parent* and sends a WELCOME_ACK message to the *parent* node

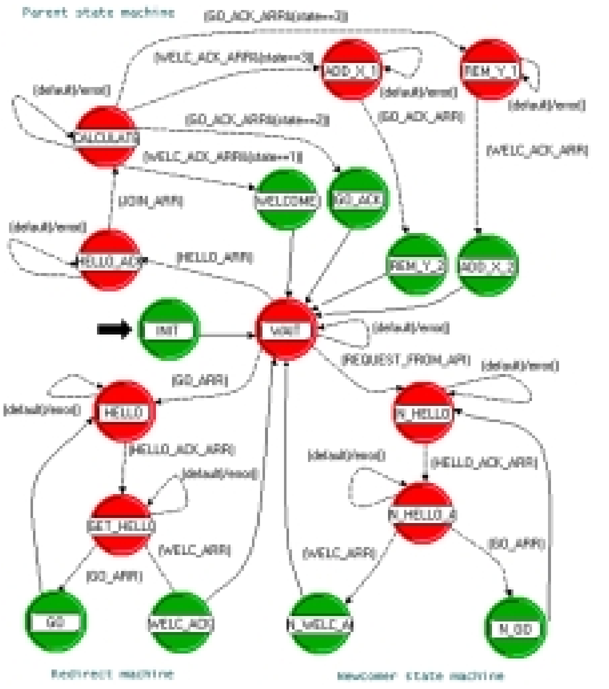


Fig. 2 The *TBCP* process model

2/ *The parent state machine:* it accepts a *newcomer* or redirects the node to one of the *parent children* and consists of eight states:

- HALLO_ACK – after receiving a HELLO message, the *parent* sends back to the *newcomer* (or to the redirected child) a HELLO_ACK message with a list of its *children*
- CALCULATE – after receiving a JOIN message with RTT table; it calculates the score function (SF) and makes a decision, where the *newcomer* will be placed; subsequent messages are send from this node depending on the result of the score function (SF):
- SF=1 - the *newcomer* goes directly under the *candidate parent*
- - the *candidate parent* sends a WELCOME message to the *newcomer* (see Fig. 1 (d1))
- SF=2 - the *newcomer* is redirected under one *candidate parent*'s child
  - the *candidate parent* sends a GO message (with address of the new *candidate parent*) to the *newcomer* (see Fig. 1 (d2))

- SF=3
  - the *newcomer* goes directly under the *candidate parent* and one of the *parent children* is redirected under the *newcomer*
  - the *candidate parent* sends a GO message to the redirected *child* and a WELCOME message to the *newcomer* (see Fig. 1 (d3))
- WELCOME – after receiving a WELCOME_ACK message from the *newcomer*, the *parent* saves the *newcomer* as a new *child* and goes to the WAIT state
- GO_ACK – after receiving a GO_ACK message either from the redirected *child* (the *parent* discards the *child* from the list of *children*) or from *newcomer* it goes to the state WAIT
- ADD_X_1 and ADD_X_2 - after receiving a WELCOME_ACK message, the *parent* saves the *newcomer* as its new *child*
- REM_Y_2 and REM_Y_1 – after receiving a GO_ACK from a redirected *child*, the *parent* removes the redirected *child* from the list of *children*

3/ *The redirect machine*: it redirects one *child* under other node, this part of TBCP state machine is similar to the *newcomer* state machine described above and consists of four state too:

- HELLO to Y – after receiving a GO message from its *parent*; the redirected *child* sends a HELLO message to the new *candidate parent*
- GET_HELLO_ACK – after receiving a HELLO_ACK message with a list of *children* from the new *candidate parent*; the *child* makes a RTT distance measurement and sends a JOIN message to the *candidate parent* with the RTT table
- GO – after receiving a GO message, which contains new *candidate parent*, the redirected *child* goes to the state HELLO
- WELCOME_ACK – after receiving a WELCOME message from the *candidate parent*, the redirected *child* saves the *candidate parent* as its new *parent*; the *child* sends a WELCOME_ACK message to the *parent* node and a GO_ACK message to the last *parent* node

The common states for all these parts are:

- INIT - variables and statistics initialisation
- WAIT – wait for event, no program code

4 Optimality of the TBCP algorithm

The TBCP is a distributed algorithm finding a Shortest Path Tree (SPT) with fanout restriction. The algorithm is not optimal mainly due to the fact that RTT values are not completely known at one node, and the solution is just “locally” optimal.

A value associated to any feasible solution is the maximum delay measured from the moment when the message is send from the *source* node to the moment when the message is delivered to the last node in the multicast group.

Static (no *newcomer*, no change of RTT values) particular instance can be represented by a graph with integer-valued length assigned to each arc. The graph vertices correspond to the nodes. The arc length between vertices *x* and *y* corresponds to the RTT between nodes *x* and *y*.

Directed graph (digraph) is constructed by assigning two oriented arcs (one from the vertex *x* to the vertex *y* and one from the vertex *y* to the vertex *x*) to each arc between vertices *x* and *y* of the underlying undirected graph. A designated vertex is a root of digraph if there are directed paths from the root to every other vertex in the digraph in the same manner, as the multicast *source* is the root of the tree spanning all members of the multicast group. The digraph is called a directed tree if it is a tree and if it contains a root. Thus if (*x*,*y*) is a (directed) arc in a directed tree, then *x* is called the *parent* of *y*, and *y* is called a *child* of *x*.

The length of the path in a digraph is the sum of the lengths of the edges on the path. The distance from *x* to *y* is defined as the length of the shortest path from *x* to *y*. Maximum delay of the multicast application using a directed tree then corresponds to *D* - the maximum distance from the root to any vertex.

The maximum distance *D* will be the only optimality criterion in the analysis of the TBCP algorithm in this article, since other possible measures of optimality (communication cost - minimum spanning tree, …) are of limited importance with respect to the practical use of multicast in applications requesting minimal response time. Algorithms minimizing the maximum distance are explained in the following text. They are indispensable in the evaluation of the results obtained by TBCP and they could be applied in the reshaping procedure foreseen for the next version of TBCP.

4.1 Finding optimal directed tree without fanout restriction

When the fanout is greater than number of outgoing arcs (*fanout(x) ≥ outbound(x))*, then the new *child y* is never rejected to join the *parent x* due to *fanout(x)* restriction. The fanout restriction looses meaning if this condition holds for all nodes taking part in the multicast communication. So the *Problem 1* can be formulated as finding a directed spanning tree with minimal *D* (maximum distance from the root) without fanout restriction. In other words the *Problem 1* is identical to the SPT problem.

Problem 1 input data:

s	*the root id (source)*
A[n,n]	*digraph adjacency matrix of lengths*
	A(i,j) - length of arc from vertex i to vertex j
	A(i,i) = 0 for all i
	A(i,j) = ∞ if there is no arc from i to vertex j

Problem 1 output data:

t[n]	*tree row vector where t(i) is parent id of vertex i*
	t[s] = 0 as root has no parent
	∀ i≠s; t[i]≠0 as each tree has n-1 arcs
d[n]	*d istance row vector specifying length of the path from the root via the spanning tree*
	maximum distance D = max(d(i)) for all i

Due to the algorithm architecture in the next section (optimistic bounding in the B&B algorithm) it makes sense to use Floyd's algorithm finding shortest paths between any pair of vertices for given digraph. Floyd's algorithm returns matrix of shortest distances *U[n,n]* and matrix of the vertex predecessors in the shortest path K*[n,n]*. It is clear that the Floyd's algorithm has time complexity O(n^3) and space complexity O(n^2) due to the efficient representation of the shortest path in the matrix P. For detailed information like 'validity of Floyd's algorithm' please refer to the graph theory textbooks [5], [3].

The *Problem 1* can be solved in polynomial time by the Floyd's algorithm:

```
Algorithm
1.Find shortest paths using Floyd's
   algorithm
2.Construct shortest path adjacency matrix
3 Extract directed tree from the shortest
   path adjacency matrix
4.Extract distance vector
```

4.2 Finding an optimal directed tree with fanout restriction

The *Problem 2* can be formulated as finding a directed spanning tree with minimal *D* (maximum distance from the *source* s) when the number of messages sent from a given node is limited by the fanout. In other words the *Problem 2* can be formulated as SPT with fanout restriction.

Problem 2 input data:

s	*the root id - as in the Problem 1*
A[n,n]	*digraph adjacency matrix of lengths - Problem 1*
f[n]	*fanout - row vector of upper bounds*
	f(i) specifies a maximum out-degree of the i-th vertex in directed spanning tree

Problem 2 output data:

t[n]	*tree row vector - as in the Problem 1*
d[n]	*distance row vector - as in the Problem 1*

The solution of the *Problem 2* adopted in this article is based on the enumeration of a finite set F of feasible solutions and the criterion *D*: F→ N with intention to find particular solution S*∈ F such that

$$D(S^*) = \min_{S \in F} D(S)$$

Enumeration methods find S* by enumeration of all S∈ F through examination of increasingly smaller subsets of F. These subsets can be treated as sets of solutions of corresponding sub-problems of the original problem.

Branch and Bound (B&B) method is one of the enumeration methods, which considers certain solutions only indirectly, without actually evaluating them explicitly. As its name implies, the B&B method consists of two fundamental procedures: branching and bounding. Branching is the procedure of partitioning a large problem into two or more sub-problems. Furthermore the sub-problems can be partitioned in similar way, etc. Bounding calculates a lower bound on the optimal solution value *D* for each sub-problem generated in the branching process.

The branching procedure can be conveniently represented as a search tree. At level 0, the search tree consists of a single partial solution (one vertex of the search tree) representing the original problem, and at further levels it consists of partial solutions representing particular sub-problems of the problem at previous level. Edges are introduced from each problem to each sub-problem. A list of partial solutions is maintained.

Suppose that at some stage of the branch and bound process a (complete) solution S_i of a criterion value $D(S_i)$ has been obtained. Suppose also that a partial solution R_j encountered in the process has an associated lover bound *optimist*(R_j). If *optimist*$(R_j) > D(S_i)$ then the partial solution R_j needs not to be considered any further in the search of *S** since resulting solution can never have a value *D* less than $D(S_i)$. When such partial solution is found, it is eliminated, since it is not needed to continue the branching process from it. The solution S_i, called a trial solution, can be found at the beginning by pursuing the tree from the top to bottom as rapidly as possible.

In order to implement the scheme of the branch and bound algorithm for the *Problem 2*, one must first decide the branching procedure and the search strategy. As illustrated in Fig. 3, very simple marking procedure could be used to find one particular path in the search tree (if the node *i* got the multicast message, then *mark(i)>0*):

```
Marking procedure
1.[Initialization] Root s has mark=1, other
   vertices have mark=0
2.[Choice of the arc] Find a set of
   candidate arcs (candidate arc A(i,j) has
   unmarked end vertex j and marked start
   vertex i with mark(i)≤f(i)), that could
   be used for the spanning tree expansion.
   Finish if such arc does not exist.
3.[Marking] Increment mark(i), increment
   mark(j) and go to the step 2.
```

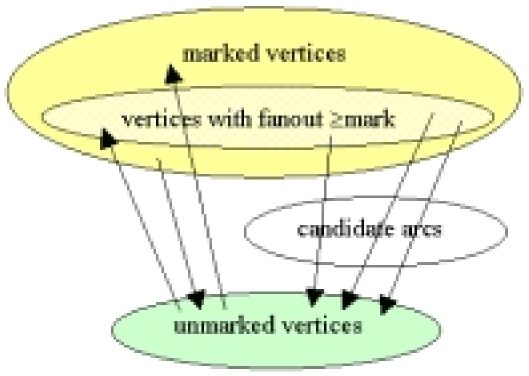


Fig. 3 Illustration of the marking procedure

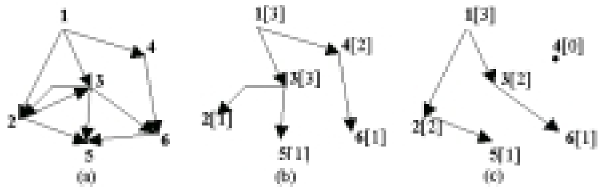


Fig. 4 Successful (a) and unsuccessful (b) result of marking procedure

In the case of instance shown in Fig. 4(a) with fanout f=[2 2 2 2 2 2] the marking procedure can finish in different ways. One can find one solution (b), where 6 vertices are marked (number of marks is indicated in []) and 5 arcs were chosen (this solution corresponds to the spanning tree [0 3 1 1 3 4]). Or one can fail (see Fig. 4(c)), as there are unmarked vertices, and there are no more *candidate* arcs (this solution corresponds to the tree [0 1 1 0 2 3] which does not spawn over all nodes).

Bounding procedure is based on two approaches:

a) **trivial bounding**

Suppose that a partial solution R_j encountered in the branching process has an associated maximum distance from the root $D(R_i)$ larger than D of the best solution known up now S_{best}. Then it is not needed to continue the branching process from R_j.

b) **optimistic bounding**

In a given step of the branching process we can make use of the matrix of shortest paths U calculated by Floyd's algorithm. Paths of our interest start at set I (set of marked vertices i with *fanout(i) ≥mark(i)*) and end at set J (set of unmarked vertices j). Moreover each marked vertex has value delay associated to it at the moment of marking and representing distance from the root. So the optimistic estimation of the distance of unmarked vertex j (corresponding to the optimistic completion time) can be calculated as *completion(j)* for all vertices from the set *J*:

$$completion(j) = \min_{i \in I}(delay(i) + U(i, j))$$

Then lover bound for *D* could be expressed as value *optimist*:

$$optimist = \max_{j \in J} completion(j)$$

So if the value optimist in some partial solution is larger than *D* of the best solution known up now S_{best}, then it is not needed to continue the branching process from this partial solution. Please note that this bounding incorporates also rejection of partial solutions with isolated unmarked vertices. For implementation details please refer to [9] or contact the authors to get Matlab source code.

4.3 Discussion

When using branch and bound algorithm one should consider the compromise between the length of the branching process and time overhead concerned with computing lower bounds or trial solutions. However, the actual computational behaviour of B&B algorithm remains unpredictable and large computational experiments are necessary to recognize their quality. It is obvious that the computational complexity of B&B algorithm is exponential in problem size when we search for optimal solution. However, the approach could be used for finding sub-optimal solutions, and then we can obtain polynomial time complexity by stopping the branching process at a certain stage or after a certain time elapsed.

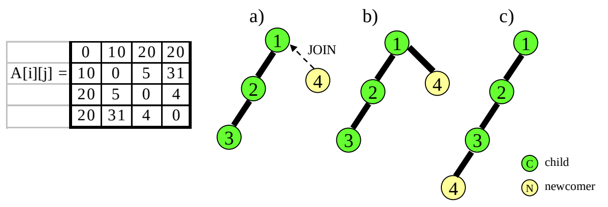
From the practical point of view there are more important issues than the time complexity of the B&B algorithm. These issues are related to the validity of the input data. Important question is how to fill the adjacency matrix A corresponding to RTT between separate nodes? First - as RTT vary in a time it is probably needed to use some statistical data. Second - due to the Internet nature the matrix A could be fully dense (with no ∞ at any entry). However from the practical point of view it needs not to be fully dense as one can make use of the geographic distance reflected partially e.g. in domains.

A very interesting theoretical issue is related to more general view of the multicast communication with replays from the multicast group (sometimes called gossiping or group multicast). In such case each node taking part in the multicast can become a root, so each node potentially needs its own directed spanning tree and optimization should be done with respect to that. Here one can make use of Floyd's algorithm finding shortest paths between any pair of vertices for given digraph in a similar way as routing protocols as OSPF use shortest path algorithms.

5 Simulation and evaluation

The optimal multicast routing tree is generated for configuration that consists of three nodes. The *parent* node has all needed information about network topology (RTT between all nodes). The non/optimality problem arises with configuration that consists of four nodes. We assume that the *source* node is always Node_1.

The first non-optimal configuration and simulation result are given in Fig. 5. The delays (RTT) between all nodes are given by the matrix A.



0	10	20	20
10	0	5	31
20	5	0	4
20	31	4	0

Fig. 5 Non-optimality of TBCP algorithm - example I

The generated TBCP tree is not identical with the optimal tree - compare *Fig. 5*b and *Fig. 5*c. This non-optimality difference is caused by missing RTT knowledge. The Node_4 wants to join the tree and it makes distance measurement (RTT) to the Node_1, and to the *child* of the Node_1 (to the Node_2). Therefore, it can send to the *candidate parent* only these values: A[4,1] and A[4,2] but not A[4,3]. This non-optimality appears when:

The Node_2 in the actual configuration is the *child* of the Node_1 and Node_3 is the *child* of the Node_2 (see *Fig. 5*a). This configuration occurred in the previous run of the TBCP when:

$$A[3][1]>A[2][1]+A[3][2]$$

Now the Node_4 (as a *newcomer*) wants to join the TBCP tree and RTTs among the nodes are such that the following conditions are satisfied:

$$A[4][1]<A[1][2]+A[2][3] \text{ AND}$$
$$A[4][1]>A[1][2]+A[2][3]+A[3][4]$$

For this configuration the difference between tree generated by TBCP (delay 20ms) and optimal spanning tree (19ms) is 1ms.

The second non-optimality example is illustrated in *Fig. 6*. The configuration of three nodes before the node 4 wants to join the TBCP tree is shown in Fig. 6a, the TBCP result after the JOIN procedure is shown in Fig. 6b and the optimal tree is shown in Fig. 6c.

The generated TBCP tree is not identical with the optimal tree - compare Fig. 6b) and Fig. 6a)

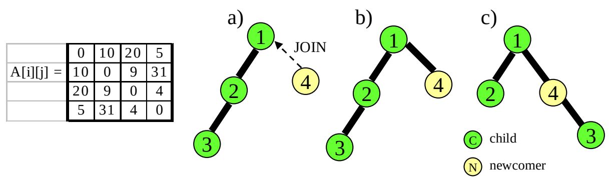
The missing RTT between Node_4 and Node_3 causes this difference. This non – optimal situation occurs if:

The Node_2 in the actual configuration is the *child* of the Node_1 and the Node_3 is the *child* of the Node_2 (see *Fig. 6* a). This configuration occurred in the previous run of the TBCP when:

$$A[3][1]>A[2][1]+A[3][2]$$

Now the Node_4 wants to join the TBCP tree and RTTs among the nodes are such that the following conditions are satisfied:

$$A[4][1]+A[4][3]<A[1][2]+A[2][3] \text{ AND}$$
$$\text{A[4][1]<A[2][1]+A[4][2]}$$



0	10	20	5
10	0	9	31
20	9	0	4
5	31	4	0

Fig. 6 Non-optimality of TBCP algorithm – example II

Delay of this solution generated by TBCP is 19ms and the optimal spanning tree has delay only 10ms.

If this situation occurs during the building of a multicast routing tree, the constructed tree isn't optimal. Please notice there are no fanout restriction in these two examples.

The example of configuration with fanout restriction (nine nodes, matrix A and fanout) is given in Fig. 7. The fanout restriction is applied for Node_1,2,3 and 4. The fanout for other nodes is so high, that it does not have any sense.

	0	8	11	12	10	20	10	10	33
A[i,,j] =	8	0	20	20	20	44	20	5	20
	11	20	0	30	30	30	30	30	30
	12	20	30	0	40	17	40	15	40
	10	20	30	40	0	50	42	50	50
	20	44	30	17	50	0	21	60	19
	10	20	30	40	42	21	0	70	70
	10	5	30	15	50	60	70	0	80
	33	20	30	40	50	19	70	80	0
fanout =	2	2	1	1	30	30	30	30	30

Fig. 7 The matrix A – configuration of nine nodes

The generated TBCP tree is shown in Fig. 8. The notation e.g. 6/2 means the node and delay (Node_6/delay 2ms). The Node_1 is always the *source* node.

Optimal spanning tree is shown in the Fig. 9. The delay differences are caused by non optimal behaviour of TBCP algorithm as was explained above (configuration of 4 nodes). The tree generated by TBCP algorithm has maximum delay 88ms (see Fig. 8) and SPT has only 40ms (see Fig. 9).

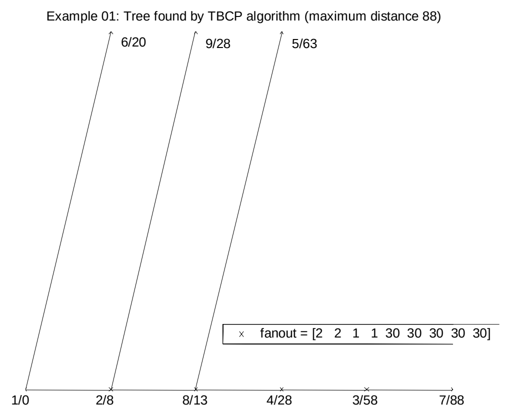

Example 01: Tree found by TBCP algorithm (maximum distance 88)
6/20
9/28
5/63
fanout = [2 2 1 1 30 30 30 30 30]
1/0
2/8
8/13
4/28
3/58
7/88

Fig. 8 Generated TBCP tree with fanout restrictions

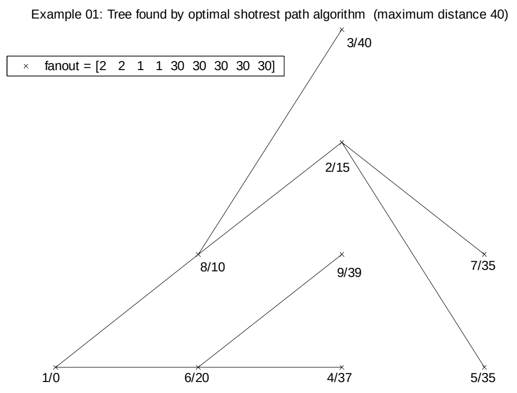

Example 01: Tree found by optimal shotrest path algorithm (maximum distance 40)
fanout = [2 2 1 1 30 30 30 30 30]
3/40
2/15
8/10
9/39
7/35
1/0
6/20
4/37
5/35

Fig. 9 Optimal spanning tree

6 Conclusions

This document explains the simulation of the TBCP protocol – the distributed algorithm finding the SPT with fanout restriction. The protocol model in OPNET allows easy maintenance of the distributed application.

Important part of this text is devoted to the optimality evaluation of the TBCP protocol. Two non-distributed algorithms (without and with fanout restriction) have been studied in order to obtain the SPT. These algorithms can serve as a base for future enhancements of the TBCP protocol. As shown in the simulation examples, the TBCP algorithm finds non-optimal solution already for 4 nodes without fanout restriction, when the joining node is not informed about the RTT values in the next level. This problem can be partly solved by gathering more information about RTT from other layres, but this still does not guarantee the tree optimality.

Finally it is needed to mention that the Branch and Bound centralised algorithm given in this article could be applied in the reshaping procedure of the distributed TBCP. Reshaping procedure can monitor the traffic of the multicast communication during a normal run of TBCP. It can make additional measurements of RTT especially for critical nodes and it can continuously gather RTT values from all nodes to the *source* node. Then the reshaping procedure can completely recalculate the tree locally in the *source* whenever it is needed. Interesting issue is the mechanism of switching from the old tree to the new one

7 References

[1] L. Mathy, R. Canonico, D. Hutchison, An Overlay Tree Building Control Protocol", *in Proceedings of the 3rd International COST 264 Workshop on Networked Group Communication (NGC)*, London, UK, November 2001, LNCS 2233, Springer-Verlag.
[2] J. de Rumeur, *Communication dans les reseaux de processeurs*, Masson Paris 1994.
[3] J. Demel, *Graphs* (in Czech), SNTL Prague, 1989.
[4] OPNET Modeler 8.0C. documentation, OPNET Technologies, Inc., Washington DC, 2000
[5] J.A., McHugh, *Algorithmic Graph Theory*, Prentice Hall, 1990.
[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, Cambridge, MA: MIT, 1992.
[7] P. Narváez , K.-Y. Siu , H.-Yi Tzeng, New dynamic algorithms for shortest path tree computation, *IEEE/ACM Transactions on Networking (TON),*Volume 8 Issue 6, December 2000
[8] R. Bellman, "On a routing problem," Q. *Appl. Math.*, vol. 16, pp. 87-90,1958.
[9] O.Dolejš, Z. Hanzálek, Simulation of Worst Case Scenarios in Multicast, *Deliverable Number 3.5.1,* Project IST-1999-10 504 GCAP, Prague, 2002

This work product was supported by the Ministry of Education of the Czech Republic under project LN00B096.