This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE Globecom 2010 proceedings.

Multi-Source Latency Variation Synchronization for Collaborative Applications

Abhishek Bhattacharya, Zhenyu Yang and Deng Pan
School of Computing and Information Sciences
Florida International University
Email: {abhat002, yangz, pand}@cs.fiu.edu

***Abstract*—Various distributed multi-source applications such as 3D Virtual Immersive Systems (3DVIS), 2D/3D Videoconferencing, Multi-party games, etc. require the construction of a multicast overlay through which the video blocks are streamed from source to receiver. The most general multicast model involves a single source and a set of receiver nodes also known as multicast set with a single stream to each receiver. In this paper, we are interested to investigate a different multicast model consisting of multiple sources and receivers which is quite common in 3DVIS/video conferencing applications where each node serves both as a source and a receiver. The problem of constructing a multicast overlay for multi-source is hard and we are interested to guarantee certain QoS constraints such as end-to-end latency and latency variations. Latency variation constraints are very important in collaborative applications for maintaining real-time interactivity and multi-stream synchronization. We present an efficient heuristic solution for the muti-source multicast construction problem with an economical time-complexity.**

I. Introduction

Distributed multi-source multimedia applications such as 3D Virtual Immersive environments and Multi-party games are becoming more popular. All these applications require the construction of a multicast subnetwork spanning one/multiple source and a set of receivers also known as the multicast set along with other intermediate nodes. Moreover, such applications also demand certain QoS guarantees in terms of end-to-end latency along individual paths from source to each receiver. In this paper, we also consider other QoS parameters based on latency variations in the multi-source multicast scenario. These type of QoS guarantees will help to provide synchronization among various receivers in a multi-stream system so that all are at a same pace with no one having moved far ahead or left behind in a multicast session.

Fig.1 illustrates the communication model of a multi-source collaborative environment where the sites are organized in an overlay network and participate in a collaborative virtual space. Each site is equipped with multiple 3D cameras and display units. Each 3D camera is spatially oriented to capture the same physical scene from various angles. Each 3D camera produces a video stream which is transmitted across the overlay network and then rendered to a shared virtual space for projection in display units [10]. In a collaborative application, such as tele-immersive dance [7], each 3DVIS site produces and receives multiple streams from other sites and thus it corresponds to a multi-source multicast scenario. [10] have proposed an efficient multi-stream content delivery framework

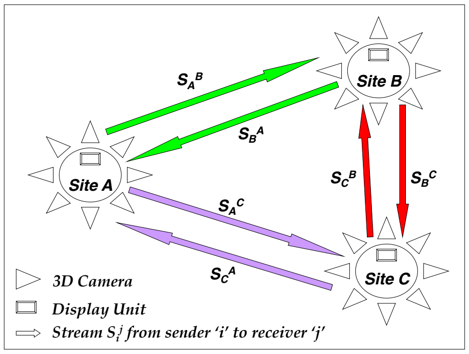


Fig. 1: A distributed Multi-Source network model

by exploiting semantic correlation among the multiple streams. As stated in [9], it is also important to reduce the latency variation among the multiple streams for synchronized delivery of video segments which is very crucial for 3D rendering and real-time interactivity of collaborative systems. For example, in Fig.1, A receives stream S_B^A from B and stream S_C^A from C. Our goal in this paper is to reduce the latency variation between S_B^A and S_C^A (*inter-source latency variation*) for A and likewise we wish to similarly achieve this for all other sites such as B and C. We would also like to have a soft-bound on the combined inter source/receiver latency variations between all streams S_i^j to keep them synchronized. We will henceforth refer to this problem as Multi-Source Latency Variation based multicast overlay or *MSLV*.

The latency variation problem was first introduced in [6] and was proved to be NP-complete. Several heuristic solutions [8], [5], [1], [9], etc followed in the literature and the problem was formulated as Delay Variation Bound Multicast Network(*DVBMN*). [9] achieved the tightest latency variation bound with a time-complexity of *O(mk log(m))* where m is the number of receivers and k is the number of shortest paths from source to each of the receivers. [2] extended the *DVBMN* problem by considering multiple paths from source to receiver and solving with the same time-complexity as [9]. Both the solutions follow a two-step framework where the first step involves in deriving k shortest paths a.k.a *ksp* from source to each receiver carried out by an efficient algorithm [4] from the literature. The next step involves the selection of optimal paths from the *ksp* list so as to achieve the minimum latency variation bound. All these solutions considered a single source in solving the problem and ignored the multi-source scenario which is more complex and the focus of this paper. Moreover, we also propose certain stringent latency variation constraints which are crucial for rendering/synchronization in a multi-source multicast overlay. We will solve the MSLV problem in an analogous two-step framework as stated above, with a new strategy for selection of optimal paths to derive the tightest latency variation.

Another alternative approach to the latency variation problem is to employ buffering at the receiver. Our argument against this approach is that the network under certain scenarios such as distributed online game applications should not rely on receivers to delay messages because it could be compromised by the end users to gain competitive edge over others. Moreover, the buffering amount is proportional to the maximum latency variation and providing bounds for this variation will result in more efficient buffer usage as stated in [6]. We believe that our scheme can be used along with receiver buffers wherever appropriate which will help to solve the latency variation problem. The remaining paper is organized as follows: The formal description of MSLV problem is introduced in Section II. In Section III we discuss our heuristic solution and algorithms for MSLV with time complexities. Experiments and Analysis are discussed in Section IV and we conclude in Section V.

II. Problem Formulation

We consider a weighted graph $G = (V, E)$ where V denotes the set of nodes, E corresponds to the set of links between the communication nodes and $n = |V|$ is the number of nodes and $m = |E|$, the number of arcs. We define an edge-latency function $L : E \rightarrow \mathbb{R}^+$ which assigns a nonnegative latency to each edge of the graph. A set of receiver nodes $D \subseteq V$ is defined with size $d = |D|$ for each source node. The graph also consists of multiple source nodes represented by a set $S \subseteq V$ of size $s = |S|$. The goal is to find a spanning forest, $\mathbb{F} = \{\mathbb{T}_i | \mathbb{T}_i = (\mathbb{V}_j, \mathbb{E}) s.t. \mathbb{E} \subseteq E \wedge \mathbb{V}_j \subseteq V; \forall 1 \leq i \leq s; \forall 1 \leq j \leq d\}$, such that each $\mathbb{T}_i$ is a subgraph that spans between a source node, $s_i\{s_i \in S\}$ and every vertex $d_j \in D$ subject to a few constraints described later in this section. We also denote $P(s_i, d_i)$ to be the path from source s_i to receiver $d_i \in D$ and the accumulative latency along this path will be $\sum_{e \in P(s_i,d_i)} L(e)$. We formulate three latency metrics as follows: (1)*End-to-End latency tolerance*, Δ, (2)*Inter-Receiver latency variation tolerance*, δ, and (3)*Inter-Source latency variation tolerance*, λ

$$\forall d_i \in D; \forall s_i \in S : \quad \sum_{e \in P(s_i,d_i)} L(e) \leq \Delta \tag{1}$$

$$\forall d_i, d_j \in D; i \neq j; \forall s_k \in S : \left| \sum_{e \in P(s_k,d_j)} L(e) - \sum_{e \in P(s_k,d_i)} L(e) \right| \leq \delta \tag{2}$$

$$\forall s_i, s_j \in S; i \neq j; \forall d_k \in D : \left| \sum_{e \in P(s_j,d_k)} L(e) - \sum_{e \in P(s_i,d_k)} L(e) \right| \leq \lambda \tag{3}$$

However, to have a strict bound on (2) and (3) we need to define the following parameters:

$$\alpha_T = max_{\forall d_i, d_j \in M; \forall s_x, s_y \in S} \left\{ \left| \sum_{e \in P(s_j,d_y)} L(e) - \sum_{e \in P(s_i,d_x)} L(e) \right| \right\} \tag{4}$$

$$\lambda_T = max_{\forall s_i, s_j \in S; i \neq j; \forall d_k \in D} \left\{ \left| \sum_{e \in P(s_j,d_k)} L(e) - \sum_{e \in P(s_i,d_k)} L(e) \right| \right\} \tag{5}$$

After defining all the latency variation metrics, let us now try to understand each of them with respect to our model in Fig: 1. (1) refers to the delay along any of the paths, S_i^j and is bounded by a constant, Δ so that longer latency paths are discarded which does not help in real-time interactivity. (2) is a latency variation metric with respect to a source such as the difference in the latencies along the path S_A^B and S_A^C for node A, $|L(S_A^B) - L(S_A^C)|$ and similarly $|L(S_B^A) - L(S_B^C)|$ for node B. Likewise, (3) is a latency variation metric with respect to a receiver such as $|L(S_B^A) - L(S_C^A)|$ for node A, $|L(S_A^B) - L(S_C^B)|$ for node B and similarly for C. These two latency metrics are similar for the model in Fig: 1 since $L(S_j^i) = L(S_i^j)$ but they are different in the more generalized model where the set of source and receiver nodes are different and is elaborated in the next section.

There can be many different combinations of applying the above latency constraints but we wish to pursue the one which is more realistic and useful for many applications. We observe that in many applications (3DVIS, video conferencing), a soft-bound on latency among all the paths will be useful which will keep all the receivers within a synchronization window (denoted as β) so that no one moves ahead or lags behind beyond the defined bound in a multicast streaming session. In other words, this constraint i.e. a bound on (4), is denoted as α_T which is $Max(L(S_i^j)) - Min(L(S_i^j))$ and should be less than β i.e. $\beta \geq max(\alpha_T)$. Another constraint that we found to be useful in collaborative applications is (5) which imposes a tight bound among the various streams received from each source. This constraint is modified from (3) and ensures that the latency variation among the multiple streams received by each node should be minimized as much as possible i.e. $\min(\lambda_T)$. Constraint (5) thus helps to effectively achieve multi-stream synchronization at the receiver which will increase the processing speed at the renderer end of a virtual immersive system. Thus, the solution we propose for MSLV is the one which keeps a soft bound(denoted as β) on α_T in (4) while attaining the tightest inter-source variation for each receiver which translates to minimizing λ_T in (5).

III. Heuristic Solution

MSLV uses a two-step framework solution analogous to [2] which is quite simple and efficient. The initial step consist of computing *ksp* for each source($s_i \in S$) to every receiver($d_j \in D$) using the algorithm proposed in [4]. The algorithm returns a set of $d \times s$ number of *ksp*-lists, with a total of $s \times d \times k$ elements. We denote each list as S_i^j where i denotes the source node

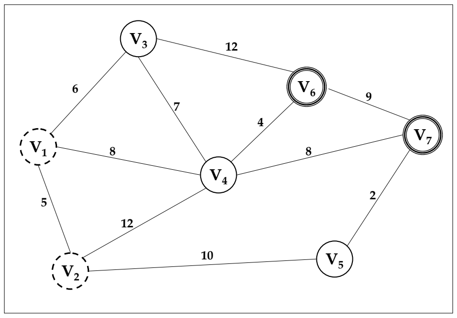
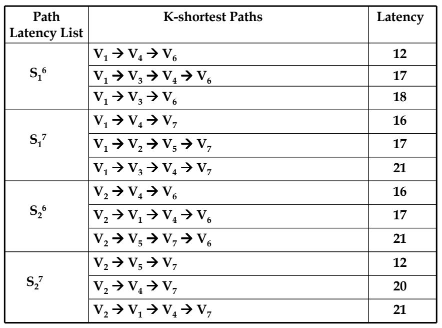


Fig. 2: Example of a graph network with link latencies.



Path Latency List	K-shortest Paths	Latency
S_1^6	$V_1 \rightarrow V_4 \rightarrow V_6$	12
	$V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6$	17
	$V_1 \rightarrow V_3 \rightarrow V_6$	18
S_1^7	$V_1 \rightarrow V_4 \rightarrow V_7$	16
	$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_7$	17
	$V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_7$	21
S_2^6	$V_2 \rightarrow V_4 \rightarrow V_6$	16
	$V_2 \rightarrow V_1 \rightarrow V_4 \rightarrow V_6$	17
	$V_2 \rightarrow V_5 \rightarrow V_7 \rightarrow V_6$	21
S_2^7	$V_2 \rightarrow V_5 \rightarrow V_7$	12
	$V_2 \rightarrow V_4 \rightarrow V_7$	20
	$V_2 \rightarrow V_1 \rightarrow V_4 \rightarrow V_7$	21

Fig. 3: Sorted latency lists in ascending order.

and j denotes the receiver node, similar to the convention of paths in Fig: 1 and will be used interchangeably which will be clear from the context. Each element of S_i^j represents a latency value that is computed as the summation of the individual latencies while traversing the path from i to j. One of the points worth noting here is that the elements in S_i^j are in ascending order which is a property of [4] and will be crucial for our algorithm. The end-to-end latency constraint in (1) can be satisfied at this stage by choosing only the values less than Δ in S_i^j and discarding the others. The second step of our framework involves the clever selection of $d{\times}s$ paths from the total $s{\times}d{\times}k$ element in *ksp* lists such that (4) is satisfied by β and (5) is minimum. MSLV reduces to the original latency variation problem in [1], [9] when $s = 1$, but we are interested in the more complex problem of multiple source in this paper.

A. *Motivational Example*

To illustrate our MSLV solution, Fig:2 shows a sample graph network where the source set is $S = \{V_1, V_2\}$, receiver set is $D = \{V_6, V_7\}$ and $\{V_3, V_4, V_5\}$ is the set of intermediate nodes. We execute the *ksp* algorithm with the graph, G in Fig: 2 as input and Δ=22 from constraint (1) with k=3. The results are listed in Fig: 3. Now our objective is to choose paths from the lists in Fig: 3 satisfying constraints (4), (5) and then merging the selected paths back to form the resulting multicast subnetwork. We know that a minimum spanning tree (MST) solution to the problem will generate the least latency paths from source to receiver but the latency variation constraint will be ignored. We define the notion of *latency-strip* denoted by $\mathbb{P}$ as a set of elements such that exactly one element is drawn from every list S_i^j. As an example, the latency-strip of MST would be $\mathbb{P}_1 = \langle 12, 16, 16, 12 \rangle$ obtained by taking the first or least elements from each of the lists $S_1^6, S_1^7, S_2^6, S_2^7$ in Fig: 3. The combined latency variation (α_T) for this latency-strip is found to be 4(16-12). If we assume $\beta = 3$, then MST solution does not qualify as a candidate for MSLV since the bound (4)($\alpha_T \leq \beta$) is not satisfied. Let us denote the inter-source latency variation for receiver i to be λ_i. For the above MST, $\lambda_6 = 4$, $\lambda_7 = 4$ and so $\lambda_T = max(\lambda_6, \lambda_7) = 4$. The MSLV solution computed by our algorithm will be the latency-strip of $\mathbb{P}_i = \langle 17, 17, 20, 20 \rangle$. By a careful analysis one can observe that $\alpha_T = 3(20 - 17)$ such that bound on (4) is satisfied since $\alpha_T \leq \beta$ and $\lambda_T = 0(\lambda_6 = 0, \lambda_7 = 0, \lambda_T = max(0, 0))$ which is the minimum latency variation in (5) from all the possible valid combinations of latency-strip. The procedure to attain this solution is discussed in Section III-C.

B. *ksp Algorithm*

The first step of MSLV solution is the computation of *ksp* from source to receiver. We use an efficient *ksp* algorithm from the literature in [4]. The Recursive Enumeration Algorithm(REA) in [4] efficiently solves a set of equations that generalize Bellman equations for the single shortest path problem. The algorithm recursively computes every new source-receiver path by visiting the nodes in the previous source-receiver path and using a heap of candidate paths associated to each node from which the next path in selected. The total time required to find the k-shortest paths in increasing order is $O(m + nk * log(m/n))$.

C. *MSLV Algorithm*

The second step of MSLV framework receives $d{\times}s$ lists as input from *ksp* algorithm with each list containing k elements signifying k-shortest paths from each source to every receiver and our objective is to achieve a latency-strip of $d{\times}s$ elements such that bound on (4) is satisfied and (5) is minimum. We can clearly observe that the brute force approach will take exponential time complexity trying for all possible $d{\times}s$ paths from $s{\times}d{\times}k$ possible paths. The design philosophy of MSLV algorithm is quite simple: we continuously try to enumerate all the possible latency-strips in each iteration by evicting the least element and adding the next element from the same list. The final output will be one of the latency-strip among all the intermediate ones with the most optimal value. Optimality in this case refers to the satisfiability of constraints (4) and (5). Since all the latency lists are sorted in increasing order, we exploit this structure to follow our iterative search. Initially, the first which is also the least element from each list, S_i^j is loaded into a data structure. Then, for each iteration, the least element from the data structure is evicted, the constraints are computed, the optimal one is updated if true and the next element from the same list is added. This iteration process continues until any of the lists are empty since by then MSLV have already exhaustively searched through all the possible combinations and found the most optimal one. The sorted list in increasing order is an important requirement for the correctness of the algorithm since we can intuitively verify that adding the next element from the sorted list at each iteration is an optimal choice since it has the highest chance to produce least variation rather than an element at a distant position. The pseudocode of MSLV algorithm is given in Table: I.

TABLE I: Pseudo-code of MSLV algorithm

```
1.  Initialize a min-heap($h$) of size $d*s$
2.  for $j \leftarrow 1$ to $d$ do
3.    for $i \leftarrow 1$ to $s$ do
4.      $h.insert(S_i^j[1])$
5.      $z_i^j \leftarrow 1$
6.  $max\_strip \leftarrow 0.0$
7.  $min\_\lambda_T \leftarrow \infty$
8.  Initialize 'd' RB-trees $(t_1, t_2, ..., t_d)$ of size 's' each
9.  for $j \leftarrow 1$ to $d$ do
10.   for $i \leftarrow$ to $s$ do
11.     $t_i.insert(S_i^j[1])$
12.     if $(S_i^j[1] > max\_strip)$
13.       $max\_strip \leftarrow S_i^j[1]$
14.     end if
15.  Initialize RB-tree '$rbt$' of size $d$
16.  Initialize $max\_latency$ array of size $d$
17.  for $i \leftarrow 1$ to $d$ do
18.    $max\_latency_i \leftarrow t_i.findmax() - t_i.findMin()$
19.    $rbt.insert(max\_latency_i)$
20.  if $(max\_strip - h.min() \leq \beta)$
21.    $min\_\lambda_T \leftarrow rbt.findMax()$
22.    $best\_\alpha_T \leftarrow max\_strip - h.min()$
23.  while($true$) do
24.    $x \leftarrow h.extractMin()$
25.    if$(x_i^j \in S_i^j)$
26.      $w \leftarrow t_i.search(S_i^j[z_i^j])$
27.      $t_i.delete(w)$
28.      $rbt.delete(max\_latency_i)$
29.      $z_i^j \leftarrow z_i^j + 1$
30.      $t_i.insert(S_i^j[z_i^j])$
31.      if$(z_i^j > k)$
32.        $break$ // out of while loop
33.      else
34.        $h.insert(S_i^j[z_i^j])$
35.        if$(S_i^j[z_i^j] > max\_strip)$
36.          $max\_strip \leftarrow S_i^j[z_i^j]$
37.        end if
38.        $max\_latency_i \leftarrow t_i.findMax() - t_i.findMin()$
39.        $rbt.insert(max\_latency_i)$
40.        if$(max\_strip - h.min() \leq \beta)$
41.          if$(rbt.findMax() < min\_\lambda_T)$
42.            $best\_\alpha_T \leftarrow max\_strip - h.min()$
43.            $min\_\lambda_T \leftarrow rbt.findMax()$
44.          end if
45.        end if
46.      end if
47.    end if
48.  end while
```

Let us try to elaborate MSLV algorithm with more details at each iteration as shown in Fig: 4. We will use the same network graph from Fig: 2 and *ksp* list in Fig: 3 for the sake of continuity with the previous discussion. The latency-

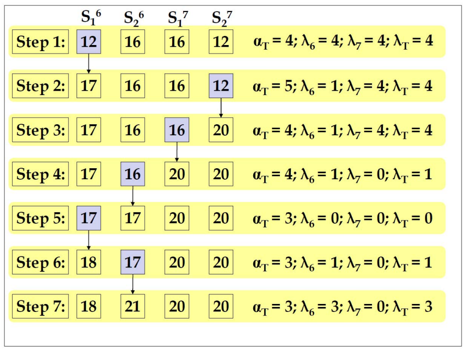

S_1^6
S_2^6
S_1^7
S_2^7
Step 1: 12 16 16 12 $\alpha_T = 4; \lambda_6 = 4; \lambda_7 = 4; \lambda_T = 4$
Step 2: 17 16 16 12 $\alpha_T = 5; \lambda_6 = 1; \lambda_7 = 4; \lambda_T = 4$
Step 3: 17 16 16 20 $\alpha_T = 4; \lambda_6 = 1; \lambda_7 = 4; \lambda_T = 4$
Step 4: 17 16 20 20 $\alpha_T = 4; \lambda_6 = 1; \lambda_7 = 0; \lambda_T = 1$
Step 5: 17 17 20 20 $\alpha_T = 3; \lambda_6 = 0; \lambda_7 = 0; \lambda_T = 0$
Step 6: 18 17 20 20 $\alpha_T = 3; \lambda_6 = 1; \lambda_7 = 0; \lambda_T = 1$
Step 7: 18 21 20 20 $\alpha_T = 3; \lambda_6 = 3; \lambda_7 = 0; \lambda_T = 3$

Fig. 4: Iterative depiction of the running algorithm

strip is always maintained in h and its contents are shown in Fig: 4 along with the *ksp* list id on the top of the figure from which the elements are taken. Before the first iteration, the contents of h are shown in the row for Step:1 and the corresponding latency variations are computed as already explained in Section: III-A. During each iteration, the least element from h is evicted and replaced with the next higher element from the same *ksp* list. For example, during iteration 1, least element 12 from S_1^6 is evicted and 17 is inserted which is the next higher element in S_1^6. We could also have evicted 12 from S_2^7 in place of S_1^6 which would only have changed the flow of the algorithm but it will produce the same result at termination and would not affect the correctness. This process is repeated in every iteration of the algorithm and will terminate whenever any of the S_i^j list is finished. The algorithm terminates after step 7 in Fig: 4 since S_1^6 is finished and no more elements are left for scanning. The result at step 5 is the optimal one and will be stored by the algorithm since $\lambda_T = 0$ is the minimum and $best\_\alpha_T = 3$. In this example, we have assumed β to be 6 and so all the latency-strips are candidates for further processing. But, if we had chosen $\beta = 4$, then the algorithm would skip Step 2 since it is not a candidate solution as constraint $\alpha_T \leq \beta$ from (4) will not satisfy.

D. *Time Complexity*

The total solution comprises of the *ksp* computation and MSLV algorithm. The time-complexity of the *ksp* algorithm is $O(m + nk * log(m/n))$ and that of MSLV is $O(dsk * log(ds))$. So, the total time-complexity of our scheme comes to $O(ms + nsk * log(m/n) + dsk * log(ds))$.

For the time complexity of MSLV algorithm, the most expensive step occurs inside the while loop and thus we analyze those section only. The most expensive steps are Line 24, 34, 40, 42 which takes $O(log(ds))$ each. Otherwise, Lines 26-27, 30, 38 take $O(log(s))$ time complexity each and Lines 28, 39, 41, 43 take $O(log(d))$ each. All the other steps in while loop are completed in $O(1)$. Now, we need to calculate the total number of iterations in while loop. Each while loop iteration removes one element and inserts another. Since there are at most $s \times d \times k$ paths to be scanned in the total *ksp*-lists, so the maximum number of possible while loop iterations is

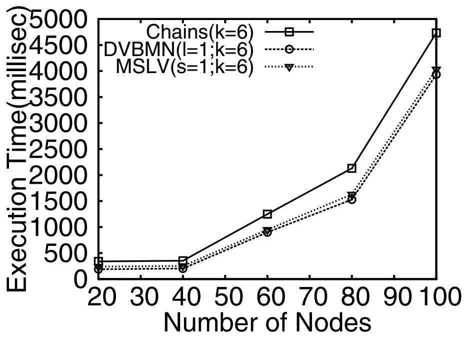


Fig. 5: Execution time comparison between Chains, DVBMN-l and MSLV

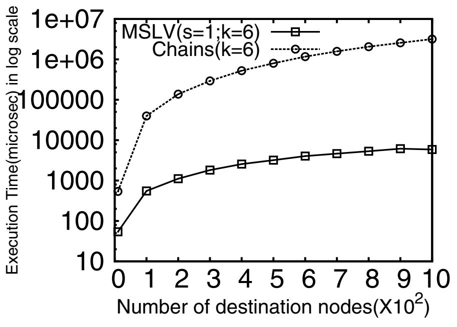


Fig. 6: Execution time comparison between Chains and MSLV

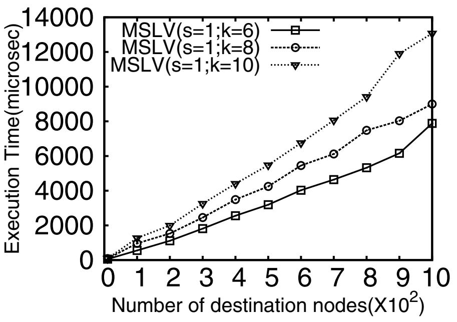


Fig. 7: Comparison with different values of k in MSLV

dsk. Therefore, time complexity of the MSLV algorithm is $O(dsk * log(ds))$.

We can observe that the MSLV algorithm does not generate the optimal combination of latency paths. We can fix this issue by recording the iteration number when the optimal latency-strip is produced in the first run and then, in the second run we can use this information to stop at the recorded step and store the corresponding latency-strip values. This will not affect the time complexity of the algorithm since there are only a constant number of runs.

IV. Experiments and Analysis

For the purpose of evaluation, we implemented Chains [1], DVBMN-l [2] and MSLV with a comparison in terms of execution time. The heuristics were run on randomly generated graphs constructed using Georgia Tech Internetwork Topology Models(GT-ITM) [3]. The topology of the overlay network were created by placing the nodes in a 2-dimensional grid of $4900kmX4900km$ to resemble the whole network spreading all over the United States.The latency for each link was set to be proportional to the Euclidean distance between the nodes connecting them. The average node degree of each graph was kept in the range of 3.5 to 4. The algorithms are tested on various graphs with the number of receivers varying from 20 to 1000. Each point in the plots represent the average value taken over 50 graphs. The experiments were performed on Intel Core 2 CPU(2.39GHz) and 2GB RAM running Linux Fedora. We have set Δ = 0.05sec for our experiments which is nearly 300 percent of the latency from source to the farthest receiver.

Since there are no algorithms that addresses the MSLV problem specifically, so to perform a fair comparison we have tested Chains, DVBMN-l and MSLV by bringing them into the same platform. We have set $l = 1$ for DVBMN-l and $s = 1$ for MSLV which makes them more or less similar in functionality to Chains. Fig: 5 shows the performance comparison graphs with 20-100 nodes and we notice that MSLV and DVBMN-l outperform Chains. The dominant cost for all these algorithms come from the *ksp* computation and though we don't see any significant difference of performance in Fig:5 but MSLV is able to solve the problem with multiple sources unlike Chains where only single source is considered. This led us to compare Chains and MSLV without *ksp* algorithm and the results are plotted in Fig:6. It is now clear from Fig:6 that MSLV performs considerably better than Chains with very high number of nodes. We can conclude that the rate of increase in the execution time of MSLV is very low as compared to Chains and so MSLV scales efficiently.

The execution time of DVBMN-l, MSLV and Chains depends largely on the value of k. We observe that the value of k increases with the increase in the number of nodes but for a fixed value of Δ, it does not require a large value of k for finding the tightest latency variation by MSLV which suggests its usefulness and feasibility in real situations. Fig: 7 plots the variation of time with increase in k for different number of receiver nodes. From Fig: 7, it can be observed that for k=10, the execution time increases almost in a linear fashion with increase of receivers. But, for lesser values of k, the rate of increase of execution time is much slower with increase in number of nodes. We choose a value of k=6 for most of our experiments.

There is an interesting relationship between Δ and k and so we need to select an optimal value of Δ to keep the problem feasible. We have experimented with different values of average node degree for each vertex in the graph and compared the values of k and Δ to achieve the tightest latency variation as shown in Fig: 8. By setting Δ to be in the range of 100%-300% (0.01-0.06sec) of the latency from the source to the farthest receiver, we observe from Fig: 8 that as the avg. node degree increases, the graph becomes more dense and so

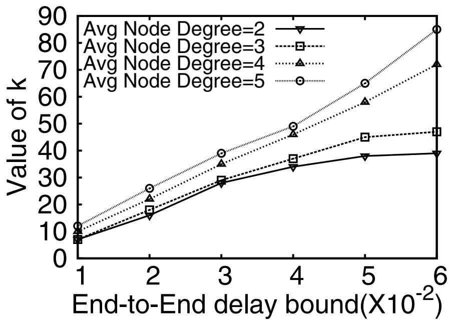


Fig. 8: k vs. Delta for varying Avg. Node Degree

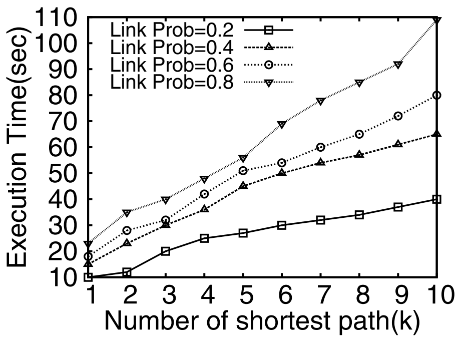
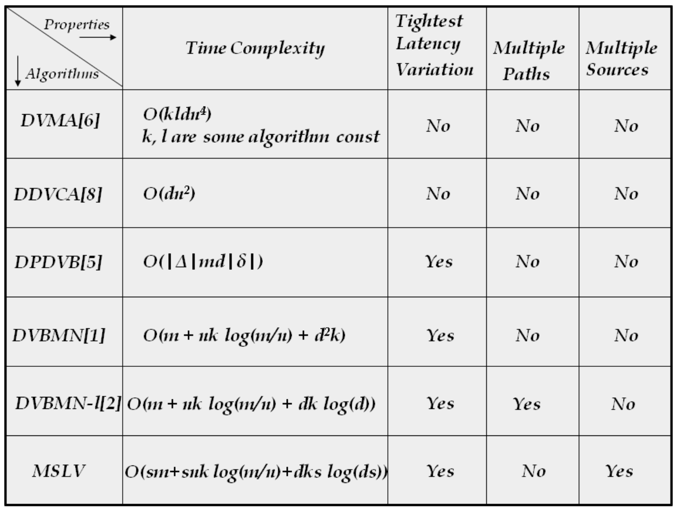


Fig. 9: Execution time for finding *ksp* with d=500 nodes



Properties → / ↓ Algorithms	Time Complexity	Tightest Latency Variation	Multiple Paths	Multiple Sources
DVMA[6]	$O(kldn^4)$ k, l are some algorithm const	No	No	No
DDVCA[8]	$O(dn^2)$	No	No	No
DPDVB[5]	$O(\vert\Delta\vert md\vert\delta\vert)$	Yes	No	No
DVBMN[1]	$O(m + nk\ log(m/n) + d^2k)$	Yes	No	No
DVBMN-l[2]	$O(m + nk\ log(m/n) + dk\ log(d))$	Yes	Yes	No
MSLV	$O(sm+snk\ log(m/n)+dks\ log(ds))$	Yes	No	Yes

Fig. 10: Analytical Comparison of the different algorithms

we need a higher value of k to achieve the tightest latency variation. So we can say that by increasing node degree or Δ, k increases since the search space increases. This led us to select an efficient *ksp* algorithm as proposed in [4]for a large network.

We also experimented the performance of *ksp* algorithm by varying the link probabilities of the generated random graphs from [3] with 500 nodes. The results are shown in Fig:9 with different values of k for each random generated graph. From Fig:9, we observe that the time to calculate 10 shortest paths with 500 nodes is 109 seconds which seems to be quite feasible. We have also performed an analytical comparison of some algorithms from the literature with respect to certain criteria which portrays the characteristic of each as shown in Fig: 10. It can be noticed from Fig: 10 that for different multicast models there are distinct solutions, such as DVBMN-*l* is suited for multi-path scenario wheras MSLV addresses multi-source case and perform better than others.

V. Conclusion

We have studied the problem of determining a multicast subnetwork with multiple sources and multiple receivers that satisfies certain QoS guarantees such as end-to-end latency and latency variations. We presented our heuristic solution in a two-step framework of *ksp* computation followed by selection of optimal paths by MSLV algorithm. Simulations were performed to evaluate our solution and was found to perform efficiently with different experimental settings. As a part of our future work, we would like to investigate the algorithm with link capacities as time varying functions. We would also like to estimate the performance of the algorithm in real-world environments with peer and network dynamics as well as to explore decentralized solutions.

References

[1] S. M. Banik, S. Radhakrishnan, and C. N. Sekharan. Multicast routing with delay and delay variation constraints for collaborative application on overlay networks. *IEEE Trans. Parallel and Distributed Systems*, pages 421–431, 2007.

[2] A. Bhattacharya and Z. Yang. Dvbmn-l: Delay variation bounded multicast network with multiple paths. In *IEEE International Conference on Multimedia and Expo*, 2009.

[3] K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling internet topology. In *IEEE Comm Magazine*, pages 160–163, 1997.

[4] V. Jimenez and A. Marzal. Computing the k shortest paths: A new algorithm and an experimental comparison. In *Proc of the 3rd Intl Workshop on Algorithm Engineering*, pages 15–29, 1999.

[5] S. Kapoor and S. Raghavan. Improved multicast routing with delay and delay variation constraints. In *IEEE GLOBECOM*, pages 476–480, 2000.

[6] G. N. Rouskas and I. Baldine. Multicast routing with end-to-end delay and delay variation constraints. *IEEE Trans. Selected Areas in Communication*, pages 346–356, 1997.

[7] R. M. Sheppard, M. Kamali, R. Rivas, M. Tamai, Z. Yang, W. Wu, and K. Nahrstedt. Advancing interactive collaborative mediums through tele-immersive dance (ted): a symbiotic creativity and design environment for art and computer science. In *ACM Multimedia*, pages 579–588, 2008.

[8] P. R. Sheu and S. T. Chen. A fast and efficient heuristic algorithm for the delay and delay variation bound multicast tree problem. In *Proc of 15th Intl Conf Information Networking*, pages 611–618, 2001.

[9] Z. Yang. Multi-stream synchronization for 3d tele-immersive and collaborative environment. In *2nd International Conference on Immersive Telecommunications*, 2009.

[10] Z. Yang, W. Wu, K. Nahrstedt, G. Kurillo, and R. Bajcsy. Viewcast: View dissemination and management for multi-party 3d tele-immersive environments. In *Proc of the 15th annual ACM Intl Conf on Multimedia*, pages 882–891. ACM Press, 2007.