
# MASTER THESIS

**TITLE: Application-Layer Multicast Algorithms for Bounded Delay Transmissions**

**MASTER DEGREE: Master of Science in Telecommunication Engineering & Management**

**AUTHOR: Jordi Pratsevall Garcia**

**DIRECTOR: Javier Ozón Górriz**

**DATE: December 16th, 2008**

**Title:** Application-Layer Multicast Algorithms for Bounded Delay Transmissions

**Author:** Jordi Pratsevall Garcia

**Director:** Javier Ozón Górriz

**Date:** December 16th, 2008

## Overview

This work shows the design and study of a family of algorithms that solves the multicast routing problem. In this problem, a given node called root has to send information to a certain group of receiving nodes. Although the algorithm can be applied at any level of the protocol stack, this paper studies its performance in the application level. This family of algorithms provides optimal routing tables between nodes belonging to the same multicast group, in such a way that the total transmission time is minimum.

The algorithms take benefit from the delay time in the transmission of a message between one peer and another to forward the data to a third peer. Beginnig with a first algorithm, defined to send only one packet, some other algorithms has been described under certain conditions to send more than a packet with the maximum possible cadence and without congestion problems. With this purpose, we have restricted the number of times that the root may send a packet and also the maximum cadence time for the rest of the nodes. Moreover, we have applied mechanisms to guarantee full connectivity.

With the aim of evaluating the performance of the different algorithms, we have calculated theoretically a set of bounds for transmission delays. Moreover, we present a serie of simulations over a virtual network that models an IP network. Over that first network, we have defined a second network of user nodes, which has been created at application level (so we can call it overlay network). We have applied the algorithms over the overlay networks, obtaining delay times, cadence times, number of nodes with congestion problems, and routing trees.

Finally, we compare the results to check the best algorithm in any case. As expected, the fastest algorithms can usually have important congestion issues (more than a 50% of affected nodes). Moreover, the algorithm defined to avoid congestion has at most 50% bigger delay than the fastest algorithms, and hence we finally advice its application in multicast transmissions.

**Títol:** Application-Layer Multicast Algorithms for Bounded Delay Transmissions

**Autor:** Jordi Pratsevall Garcia

**Director:** Javier Ozón Górriz

**Data:** 16 de desembre de 2008

**Resum**

En aquest treball hem dissenyat i estudiat una família d'algorismes que permeten solucionar el problema de l'encaminament multicast, en què un usuari anomenat arrel ha d'enviar informació a un determinat conjunt de nodes. Tot i que aquests algorismes es poden aplicar en qualsevol nivell de la torre de protocols, en aquest document s'estudia la seva utilització al nivell d'aplicació. Aquesta família d'algorismes permet crear, de manera òptima, taules d'encaminament entre els nodes d'un mateix grup multicast, de manera que es pugui minimitzar el temps que tarda la informació en arribar a tots els membres.

Els algorismes aprofiten el retard (o latència) en la transmissió d'un missatge entre un node i un altre per reenviar el paquet cap a un tercer node. A partir d'un primer algoritme, dissenyat per enviar un únic paquet a tots els nodes de la manera més ràpida possible, s'han creat altres algorismes per adaptar-los a determinades condicions i enviar més d'un paquet amb la cadència més ràpida possible i sense riscos de congestió. Per a això, s'ha limitat el nombre de vegades que el node arrel pot enviar la informació (valor que anomenem *s*), o el temps de cadència màxim dels nodes del grup. Igualment s'han aplicat mecanismes per impedir que cap membre del grup pugui quedar desconnectat de la resta d'usuaris.

Per estudiar el comportament dels algorismes s'ha calculat analíticament una sèrie de fites pels retards de transmissió. Igualment, s'han realitzat una sèrie de simulacions en una xarxa virtual que modelitza la xarxa IP, sobre la que s'ha definit una segona xarxa *overlay* formada pels nodes d'usuari, creada al nivell d'aplicació. En aquesta xarxa hem aplicat els algorismes, obtenint els temps de retard, els temps de cadència, nombre de nodes sense risc de congestió i els arbres d'enrutament per enviar la informació.

Finalment, s'han comparat els resultats per comprovar quin és el millor algorisme en cada cas. Com era de preveure, els algorismes més ràpids poden tenir problemes greus de congestió (fins a més d'un 50% de nodes afectats). Per altra banda, l'algorisme que s'ha definit per evitar aquets problemes té un retard superior de com a màxim un 50% respecte als algorismes més ràpids, de tal manera que finalment hem aconsellat la seva aplicació en la transmissió multicast de la informació.

Als meus pares, Ramon i Catalina,
i al meu germà, Miquel;
als companys de Genaker
per tot el que aprenc amb ells;
i al meu director, Javier Ozón,
per tota l'ajuda que m'ha donat.

# INDEX

# INTRODUCTION

In recent years the number of computers connected to the Internet has grown up considerably, as well as the set of applications that can be executed over them. Very often, these applications consist in data transmission between one computer and another, or between one computer and a group of them. From the origins of computer networks, the unicast (from one computer to a single one) and the broadcast (from one computer to an undefined group of them) transmissions are available; but the transmission from one single computer to a well-defined group of receivers is an unsolved problem. A lot of applications can use this type of transmission: videoconference calls, multiplayer games, or file sharing in a P2P network.

The multicast IP is an available solution to this problem, but it also has some (and relevant) drawbacks, as it was added to the original IP specification. First, the number of multicast addresses is low, which implies a limited number of groups. And second, some multicast routing algorithms and protocols are complex and, most important, all the network equipment (like routers and switches) must understand these protocols (in other case, the equipment should be changed in order to provide a multicast service).

The solution presented in this master thesis proposes a different strategy for the problem of multicast routing. We present a family of algorithms which can be used at any level (network, link or application) thanks to their degree of abstraction. These algorithms present as main features:

- Simple implementation
- Low transmission delay
- High scalability

This family of algorithms is based in a very simple approach: from a source node (also known as root), and a set of destination nodes (which would form a multicast group), the algorithm must find the simplest way to generate the optimal paths to minimize the total delay when we send some data from the source to all receivers.

At the beginning, the algorithm was defined to send a sigle packet in a very homogeneous network. After that, more complex networks and scenarios were proposed, adding also some behavior conditions to modify the initial algorithm depending on environment parameters.

To study, test and check the advantages of this family of algorithms, we have applied the algorithms on different simulated networks. In fact, we started our project creating a virtual representation of an Internet backbone, which has transit networks (with high speed and delay) and access networks (slower, with lower delays, and connected to the transit nodes).

Then, we created the overlay network by adding some nodes (or peers) on the previous networks, connecting them to the access networks. The objective is to

simulate a real Internet scenario: for example, five friends, who connect to Internet using their ADSL connection with different operators, and that want to start a multiplayer game without any centralized game server. So, the five friends will define a multicast group, and our objective is to get the best way to transmit the data between them. Anyway, this is a simple example, as our algorithms can be used in different scenarios, like an Ethernet network, and for a lot of applications and transmission levels.

Once we have the overlay network with the peers that form the multicast group, we have applied the algorithms over it. This execution will report the routing tree, the node cadence, the time at which any node receives the packet, the number of nodes with congestion problems and the total transmission delay. We have also compared the results for the different algorithms to check the best one in each case. As expected, the fastest algorithms can usually have important congestion drawback, whereas the algorithm defined to avoid congestion has a bigger delay, though it may be considered of not great importance.

The memory of this project is structured as follows: first, we explain the multicast transmission problem and we present a possible Internet network model. The second section is a brief introduction to graph theory and the Dijkstra's algorithm, which is used to obtain the minimum path between two nodes. The third section defines the family of algorithms, beginning with the initial model, used to send only one packet. Fourth section is the mathematical analysis of the algorithm, proving theoretical values. The next section is a brief description about the applications created and used to generate the overlay graph, apply the algorithm on it, and read and process the results. Finally, the sixth section contains the parameters to generate the overlay networks, results and conclusions, and the proposal for further work.

# 1. MULTICAST TRANSMISSION AND MODELING OF IP NETWORK

## 1.1. Multicast transmission issues

In all communications networks, including computer networks, there exist some transmission topology. The more usual is point-to-point transmission, like the normal telephone communication (data stream –in this case, the voice– goes from one unique source to one known receiver), or broadcast, which consists on transmitting the data from one unique source to all the possible receivers, but without specifying the recipients (which is the case of the FM radio). Anyway, if we want to transmit data from one point to a well-defined set of receivers (that is, knowing who belongs to the network), the issue is considerably more complicated. This is known as multicast transmission, where the set of clients can receive the same stream from one single source.

The application of multicast include video conferencing, multiplayer networking games, corporate communications, distance learning and distribution of software, stock quotes and news. In the context of IP networks multicast was initially proposed to be implemented at the network layer [1], but it has not been widely deployed [2]. Multicast IP defines a *multicast group* where the clients receive the stream originated by one single source, which only sends one packet to the multicast group that is forwarded to the multicast routers and replicated at points where the paths of the distinct clients diverge. By sending only one copy of the information to the network and allowing the network intelligence replicate the packet only when necessary, bandwidth and network resources may be efficiently exploited. But there are significant drawbacks to this multicast IP, as the original IP design did not have in mind multicast transmission, and it is an “add-on” to the IPv4 protocol. For example, the range of IP addresses to create multicast groups is very limited and most of them all already reserved, we need complex algorithms (like PIM) to do multicast routing over IP network, and most importantly, the network elements (like *routers*) must know these algorithms and their messages, to understand the multicast transmission.

These reasons make multicast transmission complex and difficult before using the IPv6 protocol. Also, in multicast IP the multicast group is created at level 3 of the protocol stack (that is, the network level), so an application that needs to define and manage a multicast group (for example, a videoconference call application) will find serious difficulties to take absolute control of the multicast group.

The lack of deployment of IP multicast has led to considerable interest in alternative approaches at the application layer, using peer-to-peer architectures [3]. In an application layer multicast approach, also called *overlay multicast*, participating peers organize themselves into an overlay topology for data delivery. In this topology each edge corresponds to a unicast path between two end-systems or peers in the underlying IP network. All multicast-related

functionality is implemented by peers instead of routers, with the goal of depicting an efficient overlay network for multicast data transmission. Obviously, the application-layer multicast is not as efficient as network-layer multicast, resulting in larger delays and bandwidth consumption, an in less stability of the multicast tree.

Real-time applications are especially sensitive to delay and/or delay jitter. In multiplayer networking games the information of every player has to be delivered to the rest of the players within a time interval in order to preserve game state between all the players. In video-on-demand applications coping with delay is not as important as coping with delay jitter; that is, the real point is to preserve the cadence at which the packets arrive at the destination more than the delay at which they arrive. These two constraints are important when it comes to streaming of live events, such as concerts or sports events, because the playing can not be considerably delayed from the source and the cadence must be preserved as a normal streaming video transmission. So, it is an important practical problem to determine how to depict a multicast tree which minimizes the total multicast delay between the time at which the message starts being transmited, and the time at which it is completely received by all the nodes of the multicast group.

There are several studies and proposals for application-layer multicast. These studies are mainly focused on protocols for efficient overlay tree construction and maintenance. There are two basic approaches to the problem: *fixed nodes based overlay* and *dynamic nodes based overlay*. The first approach, such as proposed in [4] and [5], places strategically some special nodes around the whole Internet which form, when required by the applications, an overlay multicast topology. Alghough the multicast tree is quite stable and easy to maintain, this solution has some of the practical problems of Multicast IP [2]. In dynamic nodes-based overlay multicast, such as [6], [7] and [8], the group members are self-organized into an overlay multicast tree. All the multicast functions are achieved by the group members. Since in large multicast groups there is a frequent joining and leaving of nodes, the adaptation to the network state is one of the main issues that should be considered together with the scalable formation of an efficient multicast tree.

## 1.2. Modeling of IP network

As said before, the overlay network used to set the multicast transmission is created over a real network. This “real network”, when we refer to computer networks, is Internet. Since the use of real networks to study the congestion, routing and managing under different routing algorithms is practically impossible, we use analysis and simulations, as long as we can consider that the used model is a “good abstraction” of the real network.

So far, the models used to modeling computer networks usually are:

- Regular topologies, like rings, trees and stars.
- “Well-known” topologies, like ARPAnet or the backbone of NSFnet.

- Randomly generated topologies.

These three cases have obvious limitations: the regular or “well-known” topologies only represent a part of current and old networks; and random topologies do not represent a real network. This is a fact to be considered, since the performance of an algorithm can widely vary from one topology to another.

Next, we present a possible Internet modeling in a graph form. To do this, we follow the model proposed by Ellen W. Zegura, Kenneth L. Calvert and Samrat Bhattacharjee [10], and we’ll see several ways to create a simulation for IP network.

The purpose of [10] is modeling in a very realistic way the *paths* (that is, the nodes sequences) along which information travels in a transmission between any two nodes in the IP network. Nodes represent switches and routers, and edges represent the paths between interconnecting elements. The model does not include individual hosts, that is, it does not consider the terminal equipment, but merely the logical structure of the network. Also, different criteria can be applied to form the model of the network, depending on the final use.

There are different ways to do this, as seen in [10]. The first one takes the parameters of well-known networks, to modelize a real network. Another option is using random plain graphs, which does not represent a real IP network but whose simplicity makes them a good option in some network studies. These models distribute nodes randomly in the plane, and then add edges between them using various probability functions.

The last way to modelize a network is using a hierarchical model. Two models are available: N-Level and Transit-Stub. The first starts with a random connected graph, and then, recursively, the nodes are substituted by another new connected graph. The issue about the edges of the upper level connectivity can be solved in various ways, for example choosing one node of the new graph randomly and make the original link support it.

The second model, Transit-Stub, is explained next.

## 1.3. The Transit-Stub network model

In this model, first we create a random connected graph, where each node represents a complete transit domain. Then, each node is substituted by a new connected graph, which represents the backbone topology for this domain (called *transit domain*). Next, for each node in each transit domain, some connected random graphs are generated, which represents the access domains added to that node (called *stub domains*). Finally, some edges between transit-domain nodes and stub-domain nodes (or even between stub-domain and stub-domain nodes) are added. If all the generated graphs are connected graphs, the resulting graph will be a connected graph. In addition, these random-generated graphs can be created using any of the random plain graphs model.

The parameters needed to create a network using the Transit-Stub model are:

- $T$: number of transit domains,
- $N_t$: average number of nodes per transit domain,
- $K$: average number of access domains per tranit node,
- $N_s$: average number of nodes per stub domain.

With these parameters, we can use the tools (described in section 5.1) to create the backbone network. We choose this model because it is simple, and gives a good approach of what is the Internet topology. Anyway, this model only sets the backbone (that is, the network-layer interconnection devices), which is not enough for our porpouses, since we need individual hosts (or peers) to act as multicast group members. To do this, we'll use a Java application (described in section 5.2) to add these multicast members to the backbone graph, and then apply our algorithms over it.

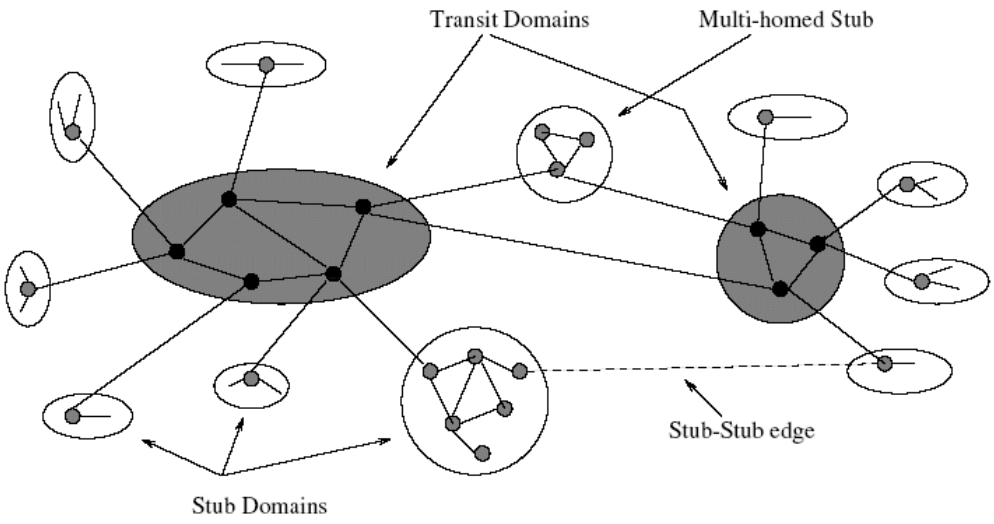



**Fig.1.1** *Transit-Stub* model

# 2. GRAPH THEORY

This section presents graph theory: some definitions about graphs and the Dijkstra's shortest path algorithm. With these tools, we will be able to modelize Internet, and create the overlay graph that will be our multicast group, and where we will apply the algorithms.

## 2.1. Definitions

A simple graph *G* is a pair *(V(G), E(G))* in which *V(G)* is a finite set of elements called *nodes* or *vertices*, and *E(G)* is a finite set of non-sorted pairs of nodes, called *arcs* or *edges*. The order *n* of a graph *G=(V,E)* is the number of nodes, or equivalently, the cardinal of *V(G)*. Also, we define the size *E* of a graph *G=(V,E)* as the number of edges in the graph, that is, the cardinal of *E(G)*. Usually, a graph is graphically represented with points as the nodes, and lines linking these points as the edges.

Either vertices and edges can have tagging functions, that is, an application $\Phi:V(G) \rightarrow Z$ and $\Phi':E(G) \rightarrow Z$ such that each single node and/or edge has an associated number (anyway, the destination set can be of any other type, like sorted-pairs of numbers or real numbers). These *tags* (also known as *weights*) can be used to identify the elements of the graph, or set some property of these elements, like link bandwidth or node type.

It is said that two nodes *u* and *v* are *adjacent* (or *neighbors*) when they are linked by an edge *uv*. In this case, nodes *u* and *v* are adjacent to edge *uv*, and edge *uv* is also said to be adjacent to nodes *u* and *v*. Two edges are adjacent when they have one common node. The degree $\delta(v)$ of a node *v* is the number of adjacent edges to *v*.

An edge *sequence* is a succession of consecutive edges $v_0v_1,v_1v_2,v_2v_3,..,v_{m-1}v_m$. This sequence draws a continuous path over the graph. A sequence with no repeated edges is called a *path*, and if there are not repeated nodes, it is called a *simple path*. A *cycle* is a path such that the start node and the end node are the same; note, however, that any node of a cycle can be chosen as the start node, and so the start is often not specified.

Two vertices *u* and *v* are *connected* if *G* contains a path from *u* to *v*. If every pair of distinct nodes in *V(G)* is connected, then the graph *G* is *connected*. A *connected component* is a maximal connected subgraph of *G*. Each node belongs to exactly one connected component, and so each edge. The *distance d(u,v)* between two nodes *u* and *v* in a graph *G* is the lenght of the shortest path between them, that is, the minimum number of edges we need to go from one node to the other. If a graph is not connected (i.e. it is *disconnected*) and two nodes belong to two different connected components, we say that their distance is infinity. The *eccentricity* $\varepsilon(v)$ of a vertex *v* in a graph *G* is the maximum distance from *v* to any other vertex of the graph. The *diameter D(G)* of a graph

*G* is the maximum eccentricity over all the vertices in the graph, and the *radius R(G)*, the minimum.

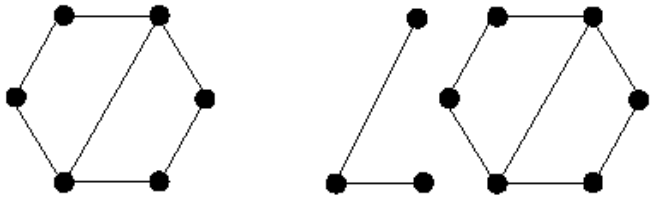


**Fig. 2.1** Examples of a connected graph and a disconnected graph

A *tree* is a connected graph with no cycles, or alternatively, a graph in which any two nodes are connected by *exactly one* path. A complete graph $K_n$ is a simple graph in which all the pairs of nodes are linked with an edge, and thus its size is *n(n-1)/2*. Moreover, if all nodes in a graph have the same degree, it's called a *regular* graph; and, in particular, if all nodes have degree *r*, it is called a regular graph of degree *r* or *r*-regular graph.

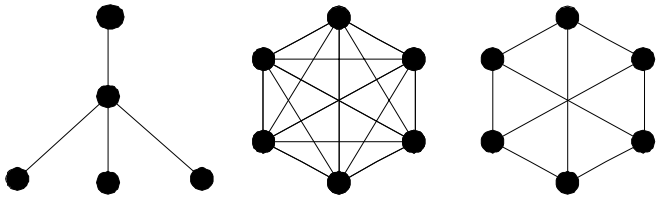


**Fig. 2.2** Examples of a tree, a complete graph, and a regular graph

A *directed graph* or *digraph G* is an ordered pair *(V(G), A(G))*, where *V(G)* is a finite, non-empty set of elements called *vertices* or *nodes*, and *A(G)* is a set of ordered pairs of vertices, called *directed edges*, *arcs* or *arrows*. An arc *e=(v,w)* (or also designed by either *(v,w)* or *vw*) is considered to be directed from *v* to *w*; *w* is called the *head* and *v* is called the *tail* of the arc. Moreover, *w* is said to be a *direct successor* of *v*, and *v* is said to be a *direct predecessor* of *w*. If a path leads from *v* to *w*, then *w* is said to be a *successor* of *v*, and *v* is said to be a *predecessor* of *w*. Note that arcs *vw* and *wv* are different, and *wv* is called the arc *vw* inverted. If *G* has not any arc from one node to itself (that is, an arc *vv* called *loop*), and all the arcs in *G* are different, then *G* is called *a simple digraph*. If *G* is a digraph, the graph obtained by deleting the "arrows" (or the directions) of the arcs is called *base graph* of *G*.

All the definitions given for a simple graph can be extended to a digraph. Thus, we can define tagging applications $\Phi:V(G) \rightarrow Z$ and $\Phi':A(G) \rightarrow Z$ over the vertex set and the arcs set, and finite sequences of arcs $v_0v_1,v_1v_2,..,v_{m-1}v_m$ where nor arcs neither vertices are repeated.

A digraph *G* is called *weakly connected* if the base graph of *G* is a connected graph. Otherwise, *G* is called *strongly connected* if it contains a directed path for every pair of nodes *v,w* $\hat{I}$ *V(G)*. While all strongly connected digraphs are connected, not all connected digraphs are strongly connected.

## 2.2. The shortest path problem. Dijkstra's algorithm

In this project, we want to study the application of some algorithms to optimize the transmission of information over a group of nodes. Later we will see in detail how these algorithms works, but now we can advance that its operation is based in the fact that computers in the same group can forward wisely the information.

To do this, first we need to find the closest nodes to a particular user, and the cost (or delay) to send them the data. This issue may arise as a simple shortest path problem, and it can be solved using the Dijkstra's algorithm, an algorithm wich finds the best path between a given node *u* of the graph and the rest of the nodes. In this algorithm, each node *v* of *G=(V,E)* has a tag *L(v)*. This tag shows the shortest known distance needed to move from one fixed node *u* to this node *v*. At the start, the value of *L(v)* is the weight *w(u,v)* of the edge that connects the nodes u and *v*, and, if this edge does not exist, this value is set to infinity (*L(v)* = ∞). Also, *L(u) = 0*, so the cost to stay in the node itself is 0. The algorithm works with a set of nodes *T* $\grave{I}$ *V*, which at every time have already obtained the shortest path from *u* to them. At the end of the algorithm execution, *L(v)* contains the cost (or delay or distance) of the shortest path to go from *u* to *v*, for any *v* in *V(G)*.

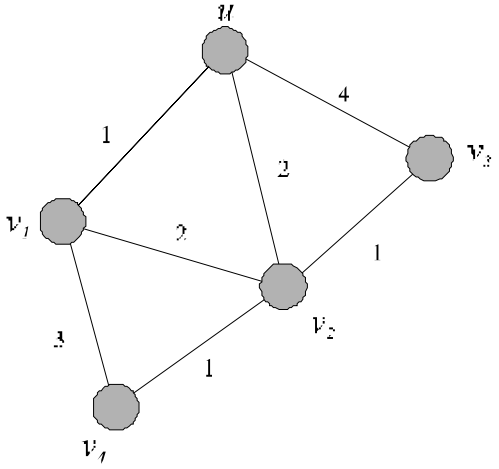
At each iteration, the algorithm adds a new node to the list *T*. This is done choosing a node *v'*, which does not belong to the list *T* and which has the minor tag *L(v')*. In other words: the algorithm chooses a node *v'* out of the list which has the minor distance from *u* to *v'*. Once this is done, the nodes supported directly by *v'* must update their tag, so the distances between *u* and these nodes are recalculated, and finally, the node *v'* is added to the list *T*. This process is repeated until all the nodes of the graph have been added to the list.

**1.** **for all** *v* ¹ *u* *L(v) = w(u,v)*
**2.** *L(u) = 0*
**3.** *T* = *{u}*
**4.** **while** *T* ¹ *V*
**init**
**5.** find *v'* Ï *T* such that ∀ *v* Ï *T* *L(v')* £ *L(v)*
**6.** *T* = *T* ∪ *{v'}*
**7.** **for all** *v* Ï *T* such that *v'* is adjacent to *v*
**if** *L(v) > L(v') + w(v',v)*
**then** *L(v) = L(v') + w(v',v)* **end if**
**end for all**
**end while**

**Fig. 2.3** Dijkstra's algorithm

Dijkstra's algorithm is optimal; to prove this, let's see that each time a node *v'* is added to *T*, the tag *L(v')* is the minimum distance from *u* to *v'*. Using a proof by contradiction, let's suppose that *L(v')* is not he shortest path between *u* and *v'*. Then, let's say that $w_2$ is the first node through which passes this new shortest way between *u* and *v'*, shorter than *L(v')*. This $w_2$ node must belong to *T* by construction, as the distance from *u* to $w_2$, which the algorithm knows since the first iteration, must be shorter than *L(v')*. The same argument can be repeated for the next node in the path, $w_3$, which must have shorter distance from *u* than *L(v')*, calculated after adding $w_2$ to *T*, and therefore this node $w_3$ must also belong to *T*. Hence, when the algorithm reaches *v'*, if it would exist a shortest path from *u* to *v'* than the indicated by the *L(v')* tag, the algorithm would have added to *T* these nodes $w_2$,$w_3$,$w_4$... which form this shortest path, and it would have found this path. Finally, as to complete the algorithm execution all nodes must be in *T*, the Dijkstra's algorithm finds the shortest path from one node *u* to any other node in the graph.

It is also easy to prove that Dijkstra's algorithm has a complexity of $O(n^2)$, something that, in practice, means that shortest paths can be found in a low computing time. To get the minimum *L(v')* (line 5 of the algorithm, Figure 2.3) we make *O(n)* comparisons, and line 7 does not need more than *n* allocations. These two lines are in the while loop from line 4, executed *(n-1)* times. Then, this algorithm can be completed in $O(n^2)$ computation time. Finally, it must be noted that this algorithm not only calculates the minimum cost (or distance) between any two nodes, but also draws the path which connects them. This can be done adding a new tag in each node, in such a way that when *L(v')* value is updated, this new tag gets the node *v* from which the new value for *L(v')* has been calculated.




| Iteration | V' | L(u) | $L(v_1)$ | $L(v_2)$ | $L(v_3)$ | $L(v_4)$ | T |
|---|---|---|---|---|---|---|---|
| 0 | - | 0 | 1 | 2 | 4 | ∞ | {u} |
| 1 | $v_1$ | 0 | 1 | 2 | 4 | 4 | {u, $v_1$} |
| 2 | $v_2$ | 0 | 1 | 2 | 3 | 3 | {u, $v_1$, $v_2$} |
| 3 | $v_3$ | 0 | 1 | 2 | 3 | 3 | {u, $v_1$, $v_2$, $v_3$} |
| 4 | $v_4$ | 0 | 1 | 2 | 3 | 3 | V |

**Fig. 2.4** Dijkstra's algorithm execution example from node *u*

# 3. ALGORITHM DEFINITION

In this section, we introduce the Postal Model and we apply it over a multicast transmission. We also prove some basic properties of the resulting algorithms and calculate their complexity. Later, in the next section, we present the mathematical analysis of the algorithms.

## 3.1. The Postal Model

To improve the data transmission between peers, Bar Noi *et al.* introduced in [9] the *MPS(n) Postal Model*, which characterizes message-passing systems which use packet switching techniques and defines a latency parameter $\lambda \geq 1$, that corresponds to the time elapsed since the message starts to be sent by one peer until it is completely received by another peer. This model is used by the authors to study the impact of communication latencies on the design of broadcasting algorithms for fully connected systems. The model considers three aspects of such systems: full-connectivity, simultaneous I/O, and communication latencies. While the two first aspects express a message-passing system as an undirected complete graph, the latter defines the same communication latency for any two nodes of the system.

In Postal Model, we use the latency of the information (that is, the time between the source has finished sending the data and the destination starts receiving it) to improve the sending rate of source. Also, the nodes or peers that have already received the information (a packet, for example) can forward it to other peers. This model searches optimum routing trees based in Fibonacci Trees, instead of the traditional binomial trees. Though the original model proposed in [9] is for a broadcast transmission, it can be extended easily to a multicast transmission, creating a well-defined group of destination peers and broadcasting the data to all of them.

Figure 3.1 presents an example of the transmission from $P_0$ to other seven more peers, using the Binomial Tree and the Fibonacci Tree. In both cases, a peer can send a packet every unit of time *t*, and the latency value is $\lambda = 2$ (that is, we need two units of time to go from one node to another). The left tree uses the Binomial Tree, which needs 6 units of time to send the data to all the peers, while the right tree uses the Fibonacci Tree from the Postal Model, and it only needs 5 units of time to complete the tree.

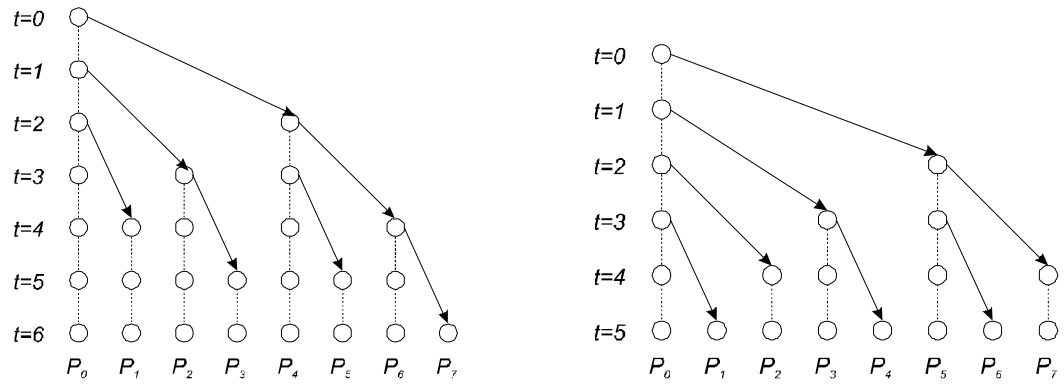



**Fig. 3.1** Transmission trees (Binomial on left, Fibonacci on right)

## 3.2. The Extended Postal Model

In this project, we extend the postal model. We define a message-passing system with $n$ peers, *EMPS(n,λ,μ)*, as a set of full-duplex peers $\{p_o,p_1,\ldots,p_{n-1}\}$ such that each peer $p$ can simultaneously send one message to peer $q$ and receive another message from peer $r$ according to the following parameters:

- For each peer $p$ in a message-passing system, we define the *transmission time* $\mu_p$ as the time that requires $p$ to transmit a message $M$ of length $l$. We denote $\mu$ the vector of all $\mu_p$'s.
- For each pair of peers $p$ and $q$ in a message-passing system, we define *communication latency* $\lambda_{pq}$ between $p$ and $q$. If at time $t$ peer $p$ starts to send a message $M$ to peer $q$, then peer $q$ sends message $M$ during the time interval $[t,\ t+\mu_p]$, and peer $q$ receives message $M$ during the time interval $[t+\lambda_{pq}-\mu_p,\ t+\lambda_{pq}]$, as shown in Figure 3.2. Note that $\lambda_{pq}$ is the sum of transmission time $\mu_p$ of peer $p$, and the propagation delay between $p$ and $q$. We denote by $\lambda$ the matrix of all $\lambda_{pq}$'s.

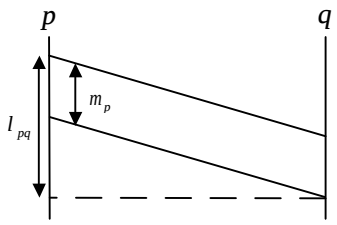



**Fig. 3.2** The Extended Postal Model

Although an overlay network will be normally fully connected (i.e. each peer in the overlay will be able to send an end-to-end message to any other peer), the performance of *EMPS(n,λ,μ)* does not require full connectivity. Hence, the definitions, bounds, and results depicted in this project will fit both fully connected and not fully connected networks. We also assume that the processing delay of the peers is negligible. However, the model could easily

take into account a peer *p* with a processing delay different to zero by adding the processing delay to the latency the first time that peer *p* forwards the message.

*EMPS(n,λ,μ)* model is a generalization of *MPS(n)* in [9]. In the latter $\mu_p$ is assumed to be the unit for all the peers, $\lambda_{pq}$ is also the same for any pair of peers *p* and *q* and, finally, the overlay network is fully connected. In *EMPS(n,λ,μ)* we consider the peers heterogeneity, and then we model different transmission times for different peers and also different communication latencies between any pair of peers, since the underlay network consists of a variable set of links and network devices.

For simplicity sake, we assume that for messages of the same length the communication latency is constant as a function of time. Hence, we do not consider the possible variation of the communication latency due to the load and broken links of the underlying network. Application-layer networks use the services provided by the underlying network, such as a TCP/IP network, to establish unicast full-duplex connections between any pair of peers. The term message refers there to any atomic piece of data sent by one peer to another using the protocols of the underlying layers.

Thus we denote by *EMPS(n,λ,μ)* the message-passing system with *n* peers, a communication latency matrix *λ* and a transmission time vector *μ*.

## 3.3. Single Message Multicast Algorithm

The problem of multicasting one message in a message-passing system is defined as follows. Let $p_0$ be a peer in *EMPS(n,λ,μ)* model which has a message *M* to multicast to the set of receiving peers $R=\{p_1,p_2,\ldots,p_{n-1}\}$ at time *t=0*, find an algorithm that minimizes the multicast time, $t_M$, that is, the time at which the last peer of *R* receives message *M*. Though the result of *EMPS(n,λ,μ)* is a multicast spanning tree (that is, a tree connecting all the peers of the network), Figure 3.3 shows that this problem is different from the well known Minimum Spanning Tree problem in which, given a network, we have to find the spanning tree with minimum weight. Furthermore, in our problem the time delay between two peers *p* and *q* is not always the weight $\lambda_{pq}$ of the edge which joins them, since if the peer *p* has forwarded the message to other peer before forwarding it to *q*, then we must add to the delay the transmission time $\mu_p$.

In [9], the authors define the algorithm BCAST which provides time-optimal multicast trees for the case of full connectivity, $\mu_p=1$ and $\lambda_{pq}=\lambda$ for any pair of peers (*p,q*). Such time-optimal multicast trees are based on *generalized Fibonacci numbers*, and they refer to these trees as *generalized Fibonacci trees*.

The authors also states in [9] that in any optimal strategy each peer, once have received message *M*, has to forward it to a new peer each time unit (recall that the transmission time is considered to be one). This idea also applies to the extended postal model *EMPS(n,λ,μ)*, with the difference that now message

retransmissions of peer *p* have to occur each transmission time $\mu_p$. The algorithm that we propose, called SMM *Single Message Multicast*, is outlined in Figure 3.4.

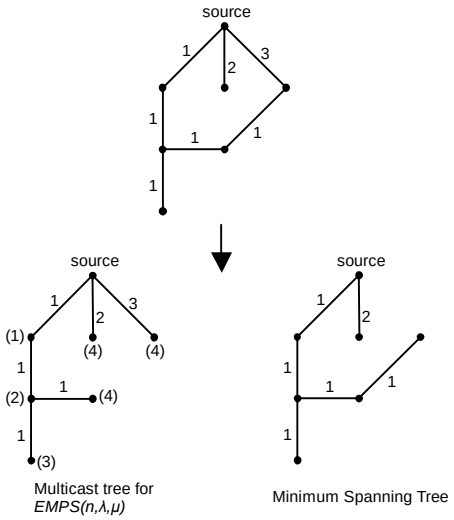



**Fig. 3.3** Comparison between *EMPS(n,λ,μ)* and Minimum Spanning Tree. In this case $\mu$=1 for all nodes and the latency is the weight of the edge. In parenthesis we write the time at which the message arrives at each node.

```
Data: EMPS(n,λ,μ)
Result: routing[i].send[j]
send ← 1;
routing[i].send[j] ← 0  ∀i, j ;
routing[i].tnext ← ∞  ∀i ;
routing[root].tnext ← lowest latency of root;
while sent < n do
    i ← imin();
    next ← routing[i].index;
    routing[i].send[next] ← 1;
    update_i(routing[i].index);
    update_t(routing[i].tnext);
    update_i(routing[next].index);
    update_tn(routing[next].tnext);
    send ← sent+1;
end
```

**Fig. 3.4** SMM Algorithm

The variables, arrays, and functions that the algorithm uses are the following:

- *routing[i].send[j]*: the routing table. Initially all its values are 0. When SMM has finished, *routing[i].send[j]* equal to 1 means that peer *i* has to forward the message to peer *j*. If it equals 0 then peer *i* will not send the message to peer *j*. As each peer has an ordered list of its neighbors according to their distance, once peer *i* has received the message, it will forward it to the first peer *j* such that *routing[i].send[j]=1*. Then it will forward the message to the next peer *k* with *routing[i].send[k]=1* and so forth.
- *i*: the peer that sends the message at each step.
- *next*: the peer which receives the message at each step.
- *routing[i].index*: points to the closest peer to *i* which has not yet received the message.
- *routing[i].tnext*: time at which if peer *i* sends a message, it will arrive at its closest peer chosen from the unvisited peers. This receiving peer is *routing[i].index*.
- *imin()*: chooses the peer with lowest *routing[i].tnext*.
- *update_i()*: searches the nearest peer to *i* from the set of peers which have not yet received the message.
- *update_t()*: once *i* has forwarded the message, *update_t()* computes the next value for *routing[i].tnext*. That is, it subtracts from its previous value the last latency, and adds up to it the next latency plus its transmission time.
- *update_tn()*: the same as *update_t()* but it applies to a peer which has just received the message. To the time at which the peer receives the message it adds up its lowest latency, chosen from the peers which have not yet received the message.

The operation of the algorithm is simple. At each step SMM chooses the peer which has not yet received the message and has the lowest cost, that is, the unvisited peer that can be reached at minimum time from any peers which has already received the message. Once the message has been received by the new peer, the algorithm recalculates the arrival times of the remaining peers (considering that the new peer can forward the message immediately), chooses the peer with the lowest arrival time and forwards the message to it. The calculations of arrival times are made under the assumption that when a sending peer finishes the retransmission of the message to another peer, it begins immediately with another destination peer. SMM algorithm is very similar to Dijkstra's shortest path algorithm [22] with the difference that in *EMPS(n,λ,μ)* the time delay between two peers *p* and *q* is not constant. Actually, in *EMPS(n,λ,μ)* this delay is equal to $\lambda_{pq}$ plus $\mu_p$ multiplied by the number of previous retransmissions of peer *p*.

Consider the network depicted in Figure 3.5 where the weights edges correspond to the communication latency $\lambda_{pq}$ between the nodes of the edge. For simplicity, we consider the transmission time up of all the peers to be equal to one. We also assume that the original sender of message *M* is peer $p_0$. At time $t$=0, $p_0$ sends the message to peer $p_1$ and this message will arrive at time

$t$=10. At time $t$=1, $p_0$ has its output link free and can send the message $M$ to the next closest peer $p_2$, which will receive the message at time $t$=11. If peer $p_1$, which is closer to peer $p_2$ than $p_0$, would forward the message, it would be received in $p_2$ at time $t$=12. However, for peers $p_3$ and $p_4$, the arrival times from $p_0$ would be $t$=22 and $t$=23, whereas from $p_1$ these arrival times would be $t$=20 and $t$=21. In this case the algorithm will forward the message to peers $p_3$ and $p_4$ from peer $p_1$.

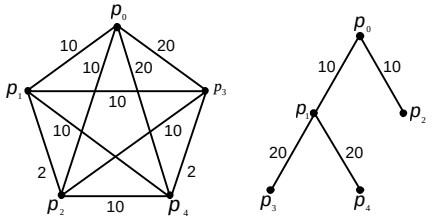



**Fig. 3.5** Example of SMM algorithm in a fully connected network

Observe that if peer $p_2$ sends the message to peer $p_4$ the message also would arrive at time $t$=21. The selection of either $p_1$ or $p_2$ for sending the message to $p_4$ depends on the use of a strict comparison in the line where the algorithm checks if the corresponding peer has to be selected (i.e. the use of "$<$" versus "$\leq$"). This consideration has an effect on the degree of the peers in the multicast tree, and also on the peers load in terms of network computing. Although the effects on computing load are not the focus of our study, it seems clearly useful to preserve the degree of the peers at the minimum. This has a certain importance since in overlay networks the peers correspond to the end-users devices.

We can prove that the multicast time achieved by the algorithm SMM is minimum when $\mu_p$=0 for all the peers. The proof is simple: in this case, since $\mu_p$=0 for all the peers, the time delay between two peers $p$ and $q$ is always the weight $\lambda_{pq}$ of the edge which joins them, and thus the SMM algorithm corresponds to the optimal Dijkstra's algorithm of complexity $O(n^2)$.

In a general case, however, the SMM algorithm is not always optimal. In Figure 3.6 we show a network where SMM is not optimal. On the left we apply SMM with a result of multicast time of 7. On the right we show that multicast delay must be reduced to 5 by redefining the order of transmission. Hence, if the source begins with peer $p_3$, follows with $p_1$ and finishes with $p_2$ the multicast delay will be 5. In the picture, we show in parenthesis the time delay for every peer. Transmission time is 1 for all the peers. On the left we apply SMM and on the right another multicast transmission order which finds a better result.

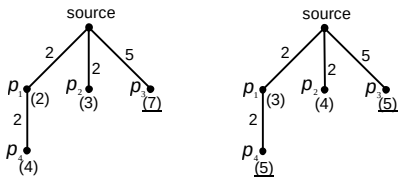



**Fig. 3.6** Example of a network where SMM is not optimal

Nevertheless, for an overlay network we can assume that $\mu_p << \lambda_{pq} \ \forall \ p,q$ and thus $\mu_p \approx 0$. This means that in an overlay network the SMM algorithm may be described as near-optimal. Moreover, in the general case where $\mu_p \neq 0$ an optimal solution could be found by means of redefining the order of transmissions for each peer. Actually, it would be possible to define for any peer all the possible orders of transmission of the message and then redefine the weight of each edge, adding the value of $i \cdot \mu_p$ to $\lambda_{pq}$, where $i$ is the order of transmission from peer $p$ to peer $q$ in each case. Then we could apply the optimal algorithm Dijkstra to all the resulting graphs and choose the best result of all. However, if we had a fully connected graph of $n$ peers, we would have $(n-1)!$ different orders of transmission for any peer which would give us, in combination with the other peers, a total number of graphs of $((n-1)!)^n$, which is not solvable in practice.

At this point, we can prove that SMM algorithm for *EMPS(n,λ,μ)* has complexity $O(n^2)$. At each step, SMM searches the peer which has not yet received the message and has lowest cost, that is, the peer that can be reached at minimum time. As the maximum number of unvisited peers is $n$ this operation requires at most $n$ comparisons. Moreover, the algorithm executes one step for every peer which receives the message and thus the total number of steps equals the number of peers. Thus, we have $n$ steps and at each step we perform at maximum $n$ comparisons plus some basic and bounded operations resulting a complexity of $O(n^2)$.

## 3.4. Message Stream Multicast Algorithm

The SMM algorithm has been defined for the multicast of a single message. This situation is not very practical and would only apply to situations where the time difference between two consecutive messages is larger than the multicast delay. If we consider a real application-layer multicast as video streaming, we see that what we call message in *EMPS(n,λ,μ)* may be a video frame (anyway, it will result on a different number of network messages and bytes to be transmitted depending on the transport protocol and the video codec used). In this case, two consecutive video frames are provided with a time difference equal to the inverse of the frames per second (fps) rate, which usually has a value between 16 and 30, depending on the video quality, and thus the source must multicast two distinct messages with an interval value between 33 and 62.5 milliseconds.

The first approach to message streaming is to repeat indefinitely the routing table obtained with SMM algorithm, multicasting each message as if it would be completely independent of the others. That means that when one message finally arrives at all the group members, the message source would proceed to multicast the next message, and so forth. The total delay multicast time of the stream $\tau_M$ would be in this case the total number of messages $M$ multiplied by the multicast SMM delay for one single message. The main inconvenience of this solution is that the source can not send the next message until the previous one has been received by all the group members and this can break the cadence of the message. In the case of video transmission, the loss of cadence can be solved by using data buffering techniques just as in video streaming to smooth delay jitter effects.

Next, we consider a new possibility. Before the first message has arrived to all peers, the source could stop sending it and begin with the second message. With this restriction, the multicast time of the first message, individually considered, will be increased, but we will begin to send before the second message, and so the third, and the fourth, and so on. This saving of time between the sending of two consecutive messages will be progressively accumulated, and, if the number of messages is large enough, it will compensate the increase of the multicast time for one single message.

The modified algorithm, that we call MSM-s *Message Stream Multicast*, will stop the transmission of the message once a peer has already sent the message $s$ times. Then it will begin to send the next message and so forth. The algorithm will apply the same multicast scheme for every message. First, it will send the first message, next the second and so forth, with the particularity that it will be able to send the second message before the first has been received by all the peers. The definition of MSM-s is then the same as SMM with the restriction on the number of retransmissions for every peer, which is at maximum $s$, and with the particularity that it will be applied to successive messages. Therefore, the scheme of the algorithm in Figure 3.4 is for MSM-s exactly the same with the only difference that function *imin()* will choose the next peer within the peers which have not yet forwarded the message $s$ times or, as shown in section 3.5, within the peers that have forwarded the message during a time lower than a certain value.

The restriction on the number of retransmissions could isolate some peers if we do not have full connectivity, as shown in Figure 3.7 when $s \leq 3$. In this case the MSM-s algorithm should choose a minimum restriction number $s$ to guarantee that all the peers receive each message. Moreover, when restricting the number of transmissions for each peer, MSM-s has to take into account the cadence of the source. That is, if the source sends at most $s$ times the first message and then, after $s \cdot \mu_r$ time units, stops the transmission of the first message to begin with the second one, we must be able to assume that the source has the second message ready to forward (suppose that $\mu_r$ is the transmission time of the source). That is, we must assume that the cadence of the source is high enough tot provide a new message each $s \cdot \mu_r$ time units. Otherwise, the source would stop sending the first message before having the second one and would remain unnecessarily idle, with the consequent loss of efficiency. Therefore,

MSM-s has to choose a minimum restriction number *s* not only to avoid the isolation of the peers (as we have seen in the former paragraph) but also in order to avoid the source idleness, that is, in such a way that $s \cdot \mu_r$ is not much lower than the cadence of the messages.

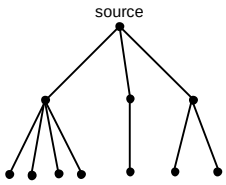



**Fig. 3.7** A network in which MSM-s isolates some peers for $s \leq 3$

For a given network, if we apply MSM-σ instead of MSM-(σ+1), that is, if the maximum number of messages sent by any peer is σ instead of σ+1, then the first message will arrive later to all the destination peers but we will usually start to send the second message before, and so the third and the fourth and so on. We have already pointed that for every new message we will save a little time. In this case, if the number of messages is large enough, the increase of the multicast time for one single message will be compensated and thus MSM-σ will be faster than MSM-(σ+1). In the next sections we prove that under certain conditions it is possible to calculate a minimum number $M\sigma$ in such a way that if the number of messages is equal or larger than $M\sigma$ then MSM-σ is better than MSM-(σ+1).

Finally, we can also prove that algorithm MSM-s for *EMPS(n,λ,μ)* has complexity $O(n^2)$. The proof is the same as for SMM, since MSM-s performs exactly the same operations with the difference that at each step MSM-s has to limit to *s* the number of message retransmissions for each peer. This little restriction, however, does not affect the complexity of MSM-s. MSM-s performs *n* steps and at each step it makes at maximum *n* comparisons plus some simple and bounded operations, including the limitations on the number of retransmissions. Thus, its complexity is $O(n^2)$.

## 3.5. MSM Algorithm with Time Restriction

In Figure 3.8 we show a peer *q* which has the same transmission time $\mu_q$ than a peer *p* which forwards to *q* the message. Let $s_p(s) \leq s$ be the actual number of times that *p* forwards the message for MSM-s. In this case the second message will be received at *q* with a delay of $s_p(s) \cdot \mu_p$ in respect to the first message, since the second message follows the same path but with a source delay of $s_p(s) \cdot \mu_p$. That is, *p* will send the first message $s_p(s)$ times and then, $s_p(s) \cdot \mu_p$ time units later, it will begin with the second and so forth. In this case, if we also limit to $s_q(s) = s_p(s)$ the number of retransmissions of *q*, then peer *q* will receive the second message at the same time that it sends the first message for the last time. Although this is not important since we have usually full-duplex

connections, it could be avoided by restricting to $s_p(s)$ the retransmissions of $p$ and to $s_q(s)=s_p(s)$-1 the number of retransmissions of peer $q$.

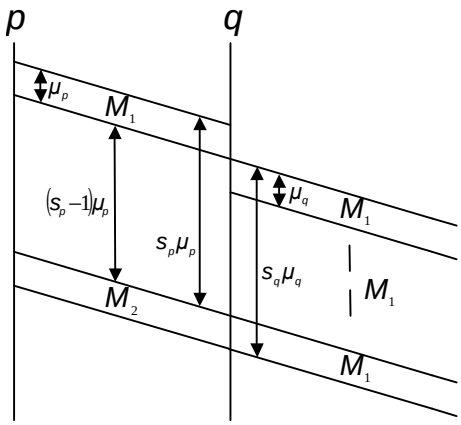



**Fig. 3.8** Transmission times of 2 different peers

In a more general case, when the forwarding period $s_q(s)\cdot\mu_q$ of peer $q$ is higher than the forwarding period $s_p(s)\cdot\mu_p$ of peer $p$ which is in a higher level of the multicast tree (i.e. which has forwarded the message in the path from the source to peer $q$), then successive messages may have higher delays than former messages. In this context, the second message could arrive at peer $q$ before it has finished forwarding the first message and then the second message would have to be buffered, with the consequent time delay. This buffering delay would be accumulated by the third message and by the forth message and so on. This situation has been avoided in the present project by limiting the time period of length $s_q(s)\cdot\mu_q$ at which each peer forwards a message, that is, by assuring that the forwarding cadence $1/(s_q(s)\cdot\mu_q)$ of any peer $q$ is higher than the cadence $1/(s_p(s)\cdot\mu_p)$ of any peer $p$ which is in the path from the source to peer $q$, including the source. Therefore, in our case the delay of the first message is the same as the time delay of any other message, an issue which has great importance in Section 4.

Furthermore, observe that we do not want $s_p(s)\cdot\mu_p >> s_q(s)\cdot\mu_q$ since in this case $q$ would stop forwarding the first message long before receiving the second one and the algorithm would lose efficiency. Thus, the value of $s_q(s)$ should be different for every peer $q$, depending on its transmission time $\mu_q$ and also on $s_p(s)$'s and $\mu_p$'s (i.e. on the parameters of the peers which depict the path that forward the message from source to peer $q$), in such a way that $s_p(s)\cdot\mu_p >\approx s_q(s)\cdot\mu_q$. Since the source is at the top of the multicast tree, we will also have for any peer $q$ the inequality $s\cdot\mu_r \geq s_r(s)\cdot\mu_r >\approx s_q(s)\cdot\mu_q$. The possibility of $s \geq s_r(s)$ is discussed in Section 4.1.

## 3.6. MSM with Cadence Restriction

Since the MSM algorithm depends on some parameters (the number of retransmissions of the root node, or the transmission time limit for any of the other peers, as seen in previous section), we can apply some modifications over the original algorithm, to improve its performance in terms of peer congestion. Thus, all the algorithms defined in the present work are based on the definition of Figure 3.4, with some changes in the *imin()* function that chooses the next forwarding peer. In fact, these changes affect mainly two different points: the condition that considers one peer better than other to be the next to forward the data; and the strategy that we follow when, under certain conditions, the algorithm can not find any node which can send the data, and the tree is still incompleted (that is, we do not have yet full connectivity). Next, we explain the algorithms that we have finally applied in last chapter.

**Cadence**: with this name, we define a MSM algorithm with time restriction, like the one defined in Section 3.5. In this case, the time limit *b0* is the value of *s* multiplied by the mean of all the root transmission times (recall that the transmission time between the root and any other peer is not always the same, since it depends on the bandwidth of all the links of the path beetwen the root and the peer). The root can send the packet at most *s* times, while the other peers will be able to send the packet while their cadence time is lower than *b0*, that is, while its total transmission time for one message is lower than *b0*. Note that, with this algorithm, depending on the value of *s* and the link bandwidth, some peers can be isolated (for example, if one peer has a very slow access, and the time to transmit the packet to it is higher than *b0*), resulting an incomplete tree. Also, we can find some cases where a node fulfills the condition for its cadence time, but this time is higher than the cadence time of the root. For example: set *s* to 5, and the value of *b0* to 50 ms (because the average $\mu$ of the root node is 10); then, if the root sends the packet to the nearest peers, whose $\mu$ is 9 for all of them, we can have a total cadence time for the root of 45 ms. As the time limit is *b0*, some other peer can have a cadence time of 48 ms, which fulfills the restriction of being lower than *b0*. Anyway, this cadence time is higher than root cadence, which could result in a congestion issue.

In this algorithm variation, the root peer could send at most *s* times the packet, instead of using the time limit value *b0*, because using the time limit may result in a more restrictive tree. To explain this, see next example: set *s*=5 and *n*=10. The root peer needs 1, 2, 2, 4, 10, 2, 2, 2, 2, and 2 ms to send the packet to the other peers, sorted by proximity (i.e. time transmission plus propagation delay). So the *b0* time limit value will be 14.5 ms. Then, if we set that the root can send *s* times the packet, we get that its cadence time may be 1+2+2+4+10=19 ms (considering that, for any of these nodes, any path between one visited node and it has a higher cost than the link from the root). If we limit the root cadence time to *b0*, we can see that once the root has sent 4 times the packet, its cadence time is 1+2+2+4=9 < *b0*, but if it sends the packet again, its cadence time will be over *b0* (19 > 14.5). Hence, the root peer would be unable to send the fifth packet.

**Forced leaves**: in this case, the root can send the packet at most *s* times, and also determines a value *b0* like the previous one. Again, the remaining nodes can send the packet while their cadence time is lower or equal to *b0*, but this time, if there is not any node that fulfills the condition of retransmission and we have already some isolated peers, the algorithm forces a leaf peer (that is, a peer that has not yet forward the packet) to send it at least once. This leaf peer is chosen with the aim of obtaining the lowest receiving time for the next isolated peer. This algorithm always generates complete trees, but now some peers not only may exceed the root cadence time, but also the time limit *b0*, resulting again in congestion issues.

**Forced all peers**: this algorithm is very similar to the previous one, but this time, if no peer fulfills the condition of retransmission and we have already isolated peers, we can force all peers that have already the message to forward it (in the previous case, we only forced the peers which had not yet forwarded the message). Again, the peer that will be forced to send the packet (and skip the conditions) is chosen in such a way that we obtain a minimum delay. The results are similar to the previous, with always complete trees, and congestion issues when some peers exceed the root cadence or the *b0* time limit.

**Dynamic**: the last algorithm does not use the *b0* time limit value. The root can send the packet at most *s* times, whereas the other peers will be able to send the packet while their cadence time is lower than the root cadence. In this case, the resulting tree may be uncompleted and the total transmission time will be usually higher than the time obtained with the former algorithms, since it is more restrictive. Nevertheless, we will not have congestion issues, since the root will have maximum cadence time and thus any other peer will finish the transmission of any message before receiving the next one.

# 4. ANALYSIS OF MSM-s

In this section we calculate theoretically some parameters of the performance of MSM-s algorithms. First, we obtain the delay transmission for one single message, and also for a generic number $M_s$ of messages, depending on the value of $s$. We also obtain an analytical bound for $M_1$, that is, for the number of messages from which the total stream multicast delay is better for $s$=1 than for $s$=2, and also an upper and a lower bound for total time delay. Finally, we provide some expressions to estimate the robustness of MSM-s algorithm.

## 4.1. Stream Multicast Delay

Let $t_s[0]$ be the time delay for an individual message when a number of transmissions is established up to $s$, $M$ the number of messages of the stream and $\mu_r$ the transmission time of the source peer, also called root. We assume that we have full connectivity, that is, $s$ is large enough to arrive at all the peers of the network. In this case, the total stream multicast delay $\tau_{Ms}$ for MSM-s is:

$$\tau_{Ms} = (M-1)\cdot s\cdot \mu_r + t_s[0] \qquad \textbf{(4.1)}$$

That is, the root sends the first message $s$ times and then, $s\cdot\mu_r$ time units later, it begins with the second and so forth. At moment $(M-1)\cdot s\cdot\mu_r$ the root finishes to send the $(M-1)$th message and it begins with the last message, that will arrive at the last peer $t_s[0]$ time units later. Remember that, as shown in Section 3.5, under certain restrictions that we consider for MSM-s the delay $t_s[0]$ for the last message is the same as the delay for any other message and so we do the calculations, for simplicity sake, for the first message. Then we denote by $t_s[0]$ the time delay for the first message.

Equation 4.1 is only valid when the root sends each message $s$ times. When the bound $s$ is large the message may be received by all peers before the root has sent it $s$ times, that is, finally the root may send the message a number of times lower than $s$. Though in this case the peer could remain idle and wait until $s\cdot\mu_r$ and then begin to send the second message, this would mean a loss of efficienty. So, for MSM-s, when the message is received for all peers before the root has sent it $s$ times, we will allow the root to send the second message immediately, without an interval of silence. In this particular case the parameter $s$ in Equation 4.1 should be replaced by the actual number of times $s_r(s)$ that the root sends each message for MSM-s, where $s_r(s) \leq s$:

$$\tau_{Ms} = (M-1)\cdot s_r(s)\cdot \mu_r + t_s[0]$$

Given the algorithm MSM-s for *EMPS(n,λ,μ)*, the delay of the first message is such that $t_{\sigma+\Delta}[0] \leq t_{\sigma}[0] \ \forall \Delta>0$. The proof of this sentence is by construction of the algorithm. When bounding up to $\sigma+\Delta$ the transmissions of each peer, MSM-$(\sigma+\Delta)$ will depict a better multicast tree than MSM-$(\sigma+\Delta-1)$ for the first message only if there exists a better solution. Otherwise, MSM-$(\sigma+\Delta)$ will depict the same multicast tree depicted by MSM-$(\sigma+\Delta-1)$. Thus $t_{\sigma+\Delta}[0] \leq t_{\sigma+\Delta-1}[0]$. Repeating the argument for $t_{\sigma+\Delta-1}[0]$ and $t_{\sigma+\Delta-2}[0]$ and so forth, we obtain $t_{\sigma+\Delta}[0] \leq t_{\sigma}[0]$ for all $\Delta>0$.

Note that this also will be valid from the second message onward if we limit the forwarding cadence $1/(s_p(s)\cdot\mu_p)$ of each peer *p* as shown in Section 3.5, with the aim of avoiding messages buffering, that is, in the case that all the messages have the same delay.

Given the algorithm MSM-s for *EMPS(n,λ,μ)*, we may obtain the conditions such that MSM-σ is faster than MSM-$(\sigma+1)$. First we define, in the case that MSM-σ could be better than MSM-$(\sigma+1)$, the minimum number $M_\sigma$ of messages from which MSM-σ is better than MSM-$(\sigma+1)$. We begin with $\sigma=1$. The value of $M_1$ can be easily obtained once we have computed $t_1[0]$, $t_2[0]$ and $s_r(2)$ by implementing MSM-1 and MSM-2. Recall that $s_r(2)$ is the number of times that the root sends each message in MSM-2 and that, as seen before, $t_1[0] \geq t_2[0]$. Then:

$$\tau_{M1} = (M-1)\cdot\mu_r + t_1[0]$$
$$\tau_{M2} = (M-1)\cdot s_r(2)\cdot\mu_r + t_2[0]$$

In this case $s_r(2)$ may be equal to either 1 or 2. In the unusual first case where $s_r(2)=1$, since $t_1[0] \geq t_2[0]$, MSM-2 will be equal or better than MSM-1 for any number of messages. In the more usual case where $s_r(2)=2$ we establish the restriction $\tau_{M1} \leq \tau_{M2}$ and then:

$$M \geq \frac{t_1[0]-t_2[0]}{\mu_r} + 1 = M_1 \qquad \textbf{(4.2)}$$

For a general case, the number $M_\sigma$ of messages from which the total stream multicast delay is better for $s=\sigma$ than for $s=\sigma+1$ may be obtained repeating the arguments for $M_1$. First we obtain the parameters of the following expressions by implementing MSM-σ and MSM-$(\sigma+1)$.

$$\tau_{M\sigma} = (M-1)\cdot s_r(\sigma)\cdot\mu_r + t_\sigma[0]$$
$$\tau_{M\sigma+1} = (M-1)\cdot s_r(\sigma+1)\cdot\mu_r + t_{\sigma+1}[0]$$

By a previous result, we have $t_\sigma[0] \geq t_{\sigma+1}[0]$. Thus, in the unusual case that $s_r(\sigma) \geq s_r(\sigma+1)$, MSM-$(\sigma+1)$ will be equal or better than MSM-$\sigma$ for any number of messages. In other case, when $s_r(\sigma) < s_r(\sigma+1)$ we may repeat the operations for $M_1$ and obtain:

$$M \geq \frac{t_\sigma[0] - t_{\sigma+1}[0]}{(s_r(\sigma+1) - s_r(\sigma)) \cdot \mu_r} + 1 = M_\sigma$$

Though it is not an usual case, in Figure 4.1 we depict a network where $s_r(\sigma)$ may be greater or equal than $s_r(\sigma+1)$. In particular, we have $s_r(3) = 3$ and $s_r(4) = 2$, if we consider that the root and peer *p* have a time transmission equal to one. Note that we could have the same case even in a fully connected network (by joining the unconnected peers of Figure 4.1 with edges of latency higher than 15) and that the situation does not depend on the forwarding cadence of peers, since in Figure 4.1 peer *p* could have, for *s*=4, either a higher or lower cadence than the root, depending on transmission times. In this case MSM-4 will be faster than MSM-3 for any number of messages.

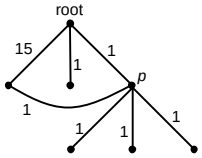



**Fig. 4.1** Example of network where $s_r(\sigma) \geq s_r(\sigma+1)$

The last result has practical consequences. As the number of transmissions allowed in MSM-s increases, we decrease the value of $t_s[0]$, but if the number of messages is large enough, we have that the total stream multicast delay may be smaller. This number $M_s$ of messages can be obtained during the process of the multicast tree computation. In consequence, the algorithm can determine the value of *s* more suitable for every performing network. Moreover, although we do not focus the subject of this project on load balancing, a small *s* may also benefit the link-stress of the network, as it will be seen in next sections.

## 4.2. Analytical bound for $M_1$

In this section we obtain an analytical bound for $M_1$, that is, for the number of messages from which the total stream multicast delay is better for $s = 1$ than for $s = 2$. As explained in former section, we assume $s_r(2) = 2$. In other case, that is, for $s_r(2) = 1$, MSM-2 will be always faster than MSM-1. We assume, as said before, that we have full connectivity both for $s = 1$ and $s = 2$.

Consider *EMPS(n,λ,μ)*, defined by one source peer, the root, and *n*-1 receivers. Let *M* be the number of messages; $\tau_{M1}$ and $\tau_{M2}$ the multicast delay for MSM-1

and MSM-2 respectively; $\mu_r$ the transmission time of the source peer (for simplicity we consider it is the same for all receiving peers); and $\lambda_{min}$ and $\lambda_{max}$ the minimum and maximum latency between any pair of peers, respectively. First, we find an upper bound for $\tau_{M1}$ and a lower bound for $\tau_{M2}$ which we denote respectively by $T_1$ and $t_2$. If we force $T_1$ to be lower or equal than $t_2$, then MSM-1 will be better than MSM-2:

$$\tau_{M1} \le T_1 \le t_2 \le \tau_{M2} \quad \textbf{(4.3)}$$

To find $T_1$ and $t_2$, we modify slightly the MSM-s performance. First we have:

$$\tau_{M1} \le (M-1)\cdot \mu_r + (n-1)\cdot \lambda_{\max} = T_1 \quad \textbf{(4.4)}$$

Recall that for MSM-1 each peer sends each message only once, so MSM-1 depicts a linear tree with $n-1$ links. In this case it is clear that $\tau_{M1} \le T_1$ since Equation 4.4 corresponds to the worst case where a message has to cross the $n$-1 links with the maximum latency $\lambda_{max}$.

To find a lower bound for $\tau_{M2}$ we consider an algorithm with a lower delay than MSM-2. First we assume that the latency for any pair of nodes is the minimum latency $\lambda_{min}$. Moreover, in the new algorithm we consider that a peer can send the same message simultaneously to two different peers, that is, that transmission time $\mu_p$ is equal to 0 for all the peers. Note that, though this is physically impossible, the new algorithm would be better than MSM-2. Let $N(t)$, $t \in Z^+$, be the number of peers that have received the message at step $t$ according to the new algorithm:

$$N(t) = \begin{cases} 1 & \text{if } t = 0 \\ 1 + 2 + 4 + \mathbf{L} + 2^t = 2^{t+1} - 1 & \text{if } t > 0 \end{cases}$$

If we equal $N(t)$ to the number $n$ of peers we will obtain the number of steps that we need to arrive at all the network:

$$t = \lceil \log_2(n+1) - 1 \rceil \quad \textbf{(4.5)}$$

In this case the new algorithm could send the single message to all the other peers in $\lceil \log_2(n+1)-1 \rceil \cdot \lambda_{min}$ time units and then for all the messages we have:

$$\tau_{M2} \ge t_2 = (M-1)\cdot 2 \cdot \mu_r + \lceil \log_2(n+1) - 1 \rceil \cdot \lambda_{\min} \quad \textbf{(4.6)}$$

Finally, if according to Equation 4.3 we force $T_1$ to be lower or equal than $t_2$ then $\tau_{M1}$ will be also lower or equal than $\tau_{M2}$ and MSM-1 will be better than MSM-2. From Eq. 4.4 and Eq. 4.6 we have:

$$M \geq \frac{(n-1)\cdot \lambda_{\max} - \lceil \log_2(n+1) - 1 \rceil \cdot \lambda_{\min}}{\mu_r} + 1$$

And since we have considered tighter cases than MSM-1 and MSM-2:

$$M_1 \leq \frac{(n-1)\cdot \lambda_{\max} - \lceil \log_2(n+1) - 1 \rceil \cdot \lambda_{\min}}{\mu_r} + 1 \qquad \textbf{(4.7)}$$

Note that when $s_r(2) = 2$ there is always a number of messages from which MSM-1 is better than MSM-2. From Equation 4.7 we see that this minimum number of messages is linear respect to the number $n$ of peers. So we can conclude that for the general case that $s_r(2) = 2$, MSM-1 is in general better than MSM-2, provided that the number of messages is usually larger than the number of peers.

## 4.3. A tighter bound for $M_1$

The bound obtained in Equation 4.7 can be improved by recalculating $t_2$, that is, by comparing MSM-2 to a tighter algorithm and by using the same lower bound $T_1$ for MSM-1. We assume that $s_r(2) = 2$. In Figure 4.2 we depict an algorithm such that, at each step, each peer sends the message to one peer and such that each peer can send the message only twice. We do not consider by the moment time delays. We call $N(t)$ the number of peers which have received the message at step $t$. Note that from step $t-1$ to next step $t$, only the $N(t-1) - N(t-3)$ peers which have not yet forwarded the message twice can forward it. Thus we have, at step $t$, the $N(t-1)$ peers of the last step plus the $N(t-1) - N(t-3)$ peers that have just received the message. Then:

$$N(t) = N(t-1) + (N(t-1) - N(t-3)) = 2\cdot N(t-1) - N(t-3)$$

In our case we have also $N(0) = 1$, $N(1) = 2$ and $N(3) = 4$. In Table 4.1 we see that $N(t) = F(t+3) - 1$ where $F(t)$ is the well known Fibonacci Serie for $F(0) = 0$ and $F(1) = 1$. Hence, considering $\lambda_1 = (1+\sqrt{5})/2$ and $\lambda_2 = (1-\sqrt{5})/2$ we have:

$$N(t) = F(t+3) - 1 = \frac{\lambda_1^{t+3} - \lambda_2^{t+3}}{\sqrt{5}} - 1 \qquad \textbf{(4.8)}$$

Suppose now that we have a network with $n = 19$ peers. From Figure 4.2 we see that at least we have to arrive at the fifth level, since at the fourth level we have only twelve peers. Actually, we could have arrived even at the sixth level if $5\lambda > 3\lambda + 3\mu$, but we can not assure it. Moreover, from Figure 4.2 we see that if we consider time delays we can arrive at level $t$ at minimum time of $t \cdot (\lambda_{\min} + \mu_{\min})/2$, where $\lambda_{\min}$ is the minimum latency of the network and $\mu_{\min}$ is the minimum transmission time of the peers. Remember that by definition $\lambda_{\min} \geq \mu_{\min}$.

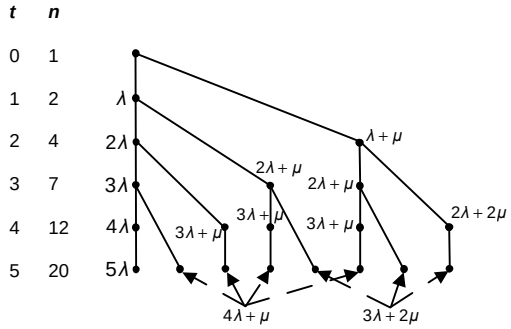



**Fig. 4.2** The Fibonacci tree

**Table 4.1.** $N(t)$ vs. Fibonacci Series

| ***t*** | | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***N(t)*** | | | | 1 | 2 | 4 | 7 | 12 | 20 | 33 |
| ***F(t)*** | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |
| ***t*** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

As we are determining a lower bound, in order to calculate the number $t$ of steps as a function of the number $n$ of peers we define $N'(t)$ which is a little faster than $N(t)$:

$$N'(t) = \frac{\lambda_1^{t+3} + 1}{\sqrt{5}} - 1$$

Observe that from Equation 4.8 we have -1 < $\lambda_2$ < 0 and then $N'(t) > N(t)$. Hence, if we calculate for $N'(t)$ the number of steps necessary to visit $n$ peers, we will obtain a lower bound for $N(t)$. For $t >> 1$ the term $\lambda_2^{t+3}$ is close to 0 and then we have a very accurate bound. If we equal $N'(t)$ to $n$ we obtain:

$$t = \left\lceil \frac{\log\left((n+1)\sqrt{5} - 1\right)}{\log \lambda_1} - 3 \right\rceil$$

And considering that at each step we have a minimum delay of $(\lambda_{\min} + \mu_{\min})/2$ then we have:

$$\tau_{M2} \geq t_2 = (M-1)\cdot 2\cdot \mu_r + \left\lceil \frac{\log\left((n+1)\sqrt{5}-1\right)}{\log \lambda_1} - 3 \right\rceil \frac{\lambda_{\min} + \mu_{\min}}{2}$$

Hence, considering the new bound of $t_2$ with $\lambda_1 = (1+\sqrt{5})/2$ and repeating the arguments from the former section with the same value of $T_1$, we have:

$$M_1 \leq \frac{(n-1)\cdot \lambda_{\max} - \left\lceil \frac{\log\left((n+1)\sqrt{5}-1\right)}{\log \lambda_1} - 3 \right\rceil \frac{\lambda_{\min} + \mu_{\min}}{2}}{\mu_r} + 1 \qquad \textbf{(4.9)}$$

This bound is tighter than the bound of Eq. 4.7 depending on the value of $\mu_{\min}$. Actually, if $\mu_{\min}$ is close to $\lambda_{\min}$ the new bound is better than the former, whereas if $\mu_{\min} << \lambda_{\min}$ then we must consider Equation 4.7. In a practical case we can calculate both bounds and consider the best one.

## 4.4. An upper bound for Time Delay in MSM-s

Let $\tau_{Ms}$ be the multicast delay for MSM-s and $T_s$ an upper bound of $\tau_{Ms}$. We obtain first a bound for $s = 2$ and then we generalize the result. To calculate $T_2$ we consider once more the algorithm of Figure 4.2. Suppose again that we have a network with $n = 19$ peers. We have to arrive at least at the fifth level, since at the fourth level the message has only arrived at twelve peers. Actually, with 19 peers we could have arrived at even the sixth level if $5\lambda > 3\lambda + 3\mu$, but in this case the delay due to $3\lambda + 3\mu$ would be lower than the maximum delay $5\lambda$ at the fifth level. As we calculate an upper bound we will considerate the value of $5\lambda$.

As we determine an upper bound, in order to calculate the number *t* of steps as a function of the number $n$ of peers we define $N''(t)$ which is a little slower than $N(t)$:

$$N''(t) = \frac{\lambda_1^{t+3} - 1}{\sqrt{5}} - 1$$

From Equation 4.8 we have $1 < \lambda_2 < 0$ and then $N''(t) < N(t)$. If we calculate for $N''(t)$ the number of steps that we need to visit $n$ peers we will obtain an upper bound for $N(t)$. Remember that for $t >> 1$ we will have a fitted bound. If we equal $N''(t)$ to $n$ we obtain:

$$t = \left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil$$

And considering that at each step we have a maximum delay of $\lambda_{\max}$ :

$$t_2[0] \leq \left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max}$$

Finally, since $t_s[0] \leq t_2[0] \ \forall s \geq 2$ and the root sends $s_r(s)$ times each message, where $s_r(s) \leq s$, we have:

$$\begin{aligned} \tau_{Ms} &\leq (M-1) \cdot s_r(s) \cdot \mu_r + \left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} \leq \\ &\leq (M-1) \cdot s \cdot \mu_r + \left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} \quad \forall s \geq 2 \end{aligned} \qquad \textbf{(4.10)}$$

Furthermore, for $s = 1$ we apply Equation 4.4.

## 4.5. A lower bound for Time Delay in MSM-s

To find a lower bound for $\tau_{Ms}$ we consider, as we did for MSM-2 in section 4.2 but now in a general case, an algorithm such that a peer can forward the same message simultaneously to *s* different peers. Moreover, we assume that the latency for any pair of nodes is the minimum latency $\lambda_{\min}$. The new algorithm is therefore faster than MSM-s. Let $N(t), t \in Z^+$ be the number of peers that have received the message at step $t$ :

$$N(t) = \begin{cases} 1 & \text{if } t = 0 \\ 1 + s + s^2 + \mathbf{L} + s^t = \dfrac{s^{t+1}-1}{s-1} & \text{if } t > 0 \end{cases}$$

If we equal $N(t)$ to the number of peers $n$ we obtain the number of steps that we need to arrive at all the network:

$$t = \left\lceil \frac{\log(n(s-1)+1)}{\log s} - 1 \right\rceil \quad \forall s \geq 2 \qquad \textbf{(4.11)}$$

And thus:

$$\begin{aligned} \tau_{Ms} &\geq (M-1)\cdot s_r(s)\cdot \mu_r + \left\lceil \frac{\log(n(s-1)+1)}{\log s} - 1 \right\rceil \cdot \lambda_{\min} \geq \\ &\geq (M-1)\cdot \mu_r + \left\lceil \frac{\log(n(s-1)+1)}{\log s} - 1 \right\rceil \cdot \lambda_{\min} \quad \forall s \geq 2 \end{aligned} \tag{4.12}$$

Note that for $s=2$ we obtain the expression in Equation 4.6. For the case $s=1$ we obtain the following expression:

$$\tau_{M1} \geq (M-1)\cdot \mu_r + (n-1)\cdot \lambda_{\min}$$

## 4.6. A general bound for $M_\sigma$

Taking the upper and lower bounds of former sections and repeating the arguments of section 4.2 for the bound of $M_1$, we can obtain a bound for the minimum number $M_\sigma$ of messages from which MSM-σ is better than MSM-(σ+1). Remember that, as we have said in the in section 4.1, this bound has only sense when $s_r(\sigma) < s_r(\sigma+1)$. In other case, MSM-(σ+1) will be always better than MSM-σ, no matter how large is the number of messages. We consider $\sigma \geq 2$.

From Equation 4.10 the upper bound for $s=\sigma$ is:

$$\tau_{M\sigma} \leq (M-1)\cdot s_r(\sigma)\cdot \mu_r + \left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} = T_\sigma$$

And from Equation 4.12 the lower bound for $s=s+1$ is:

$$\tau_{M(\sigma+1)} \geq (M-1)\cdot s_r(\sigma+1)\cdot \mu_r + \left\lceil \frac{\log(n\sigma+1)}{\log(\sigma+1)} - 1 \right\rceil \cdot \lambda_{\min} = t_{\sigma+1}$$

Forcing $T_\sigma \leq t_{\sigma+1}$ we will have $\tau_\sigma \leq \tau_{\sigma+1}$ and MSM-σ will be better than MSM-(σ+1):

$$M \geq \frac{\left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} - \left\lceil \frac{\log(n\sigma+1)}{\log(\sigma+1)} - 1 \right\rceil \cdot \lambda_{\min}}{(s_r(\sigma+1) - s_r(\sigma))\cdot \mu_r} + 1$$

And then, since we have a pessimistic case:

$$M_\sigma \le \frac{\left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} - \left\lceil \frac{\log(n\sigma+1)}{\log(\sigma+1)} - 1 \right\rceil \cdot \lambda_{\min}}{\left(s_r(\sigma+1) - s_r(\sigma)\right) \cdot \mu_r} + 1 \le$$

$$\le \frac{\left\lceil \frac{\log\left((n+1)\sqrt{5}+1\right)}{\log \lambda_1} - 3 \right\rceil \cdot \lambda_{\max} - \left\lceil \frac{\log(n\sigma+1)}{\log(\sigma+1)} - 1 \right\rceil \cdot \lambda_{\min}}{\mu_r} + 1 \quad \forall \sigma \ge 2$$

## 4.7. Robustness of MSM-s

Frequently, real-time applications use unreliable transport-layer protocols such as User Datagram Protocol (UDP). That means that it is not always possible to ensure the ordered and complete arrival of the data at the destination peers. The overlay links of application-layer multicast for real-time applications could therefore provide some degree of reliability. We analyze in this section, under the assumption that there is no message retransmission, the robustness of MSM-s algorithm.

First note that MSM-1 algorithm depicts a linear topology for the multicast tree, that is, a message arrives at a peer which immediately forwards it to another peer and so forth, whereas the MSM-s algorithm depicts in general *s* divergent paths from each peer of the multicast tree, as shown in Figure 4.3. In this case, for MSM-1 the probability that a message arrives at $l$ peers is always lower than the probability of arriving at $l-1$ peers. This is because for arriving at the $l$th peer the message will have to arrive first until the $(l-1)$th peer and then cross successfully the edge between them. This undesirable characteristic does not appear in MSM-s when $s>1$ due to the different multicast tree that depicts the algorithm, with divergent paths. In MSM-3, for example, the probability of arriving at three peers is much higher than the probability of arriving at only one, an issue which does not happen in MSM-1. Then, as we show in this section, MSM-1 will be less robust than the rest of MSM-s algorithms.

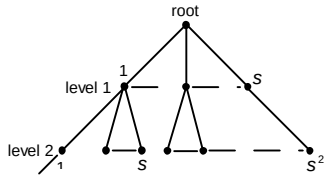



**Fig. 4.3** Multicast tree for the calculation of MSM-s roustness

Let $P_c(p,q)$ be the probability that peer $q$ receives correctly a message sent by peer $p$. For the sake of simplicity we consider that $P_c(p,q) = P_c$ for all the peers. Actually, if we consider $P_c = \max\{P_c(p,q)\} \quad \forall p,q \in EMPS(n,\lambda,\mu)$, we will get a lower bound of the robustness of the MSM-s algorithm. We call a peer which has received correctly the message a "visited peer". Note that, since the results would be the same, we calculate the robustness only for the transmission of one single message.

We denote by $\overline{n}_s$ the average number of peers that receive the message for MSM-s with a probability $P_c$ for each peer-to-peer communication. To calculate $\overline{n}_s$ we divide the peers into levels, according to Figure 4.3. We call $E_l(n)$ the average number of peers that receive the message at level $l$. For the first level we have $s$ peers and then:

$$E_1(n) = E(i_1 + i_2 + \mathbf{L} + i_s) = s \cdot E(i_1) = s \cdot (0 \cdot p(0) + 1 \cdot p(1)) = s \cdot P_c$$

Note that $i_j$ refers to the number of messages received by the $j$th peer at level $l$. This number may be 0 or 1 depending on whether the peer has received the message or not (we calculate the average number of visited peers when we send only one message). Thus $i_1 + i_2 + \cdots + i_s$ is equal to the number of peers that have received the message. For the $s^2$ peers of the second level, we have:

$$E_2(n) = E(i_1 + i_2 + \mathbf{L} + i_{s^2}) = s^2 \cdot E(i_1) = s^2 \cdot (0 \cdot p(0) + 1 \cdot p(1)) = s^2 \cdot P_c^2$$

And in general for the level $l$:

$$E_l(n) = E(i_1 + i_2 + \mathbf{L} + i_{s^l}) = s^l \cdot E(i_1) = s^l \cdot (0 \cdot p(0) + 1 \cdot p(1)) = s^l \cdot P_c^l$$

Finally we can calculate $\overline{n}_s$ as the sum of the averages of each level, considering that, since the root always will have the message, for level 0 this average is 1. We denote by the number $L$ of levels:

$$\begin{aligned} \overline{n}_s &= E_0(n) + E_1(n) + \mathbf{L} + E_L(n) = \\ &= 1 + s \cdot P_c + s^2 \cdot P_c^2 + \mathbf{L} + s^L \cdot P_c^L = \frac{(s \cdot P_c)^{L+1} - 1}{s \cdot P_c - 1} \end{aligned} \qquad \textbf{(4.13)}$$

In this case we assume that the number of peers is $1 + s + s^2 + \mathbf{L} + s^L = (s^{L+1} - 1)/(s - 1)$ and that we flood the network level by level (this only would happen on a very regular network with little time transmissions). Nevertheless we have been shown by simulations that the results from

Equation 4.13 are quite realistic in the case that $n = \left(s^{L+1} - 1\right)/\left(s - 1\right)$. For the MSM-1 algorithm the assumptions of Equation 4.13 are valid. Since we have $L$=$n$-1 it results:

$$\bar{n}_1 = \frac{1 - P_c^n}{1 - P_c} < \frac{1}{1 - P_c}$$

Thus if 1/(1-$P_c$) is much smaller than the number $n$ of peers, the average number of visited peers with MSM-1 will be also much smaller than $n$. But if, on the contrary, 1/(1-$P_c$) is higher than $n$ then the average number of visited peers may be close to $n$. In Table 4.2 we see some values of the average for MSM-1 depending on $n$ and $P_c$. For $P_c$=0.9 we have 1/(1-$P_c$)=10 and then the average may not be greater than 10, no matter how high is $n$. However, for the usual values of $P_c$=0.999 and $n = 50$ or $n = 100$ we have good averages, close to $n$. For the other values the average is much smaller than $n$, which means that the message is not received by a large percentage of peers. In this case the algorithm should automatically change from MSM-1 to MSM-2. For instance, for $n \approx 1000$ and $P_c$=0.999 we would arrive at only the 63.2% of the peers with MSM-1 whereas for MSM-2 the percentage would be of 99.1%. In this case the percentage for MSM-3 would be only a little higher than for MSM-2: 99.4%. Actually, in a general case the robustness of MSM-2 will be acceptable.

**Table 4.2.** Average of the number of peers that receive the message for MSM-1, depending on $n$ and $P_c$

| ***n*** | **$P_c$ = 0.9** | **$P_c$ = 0.99** | **$P_c$ = 0.999** |
|---|---|---|---|
| 50 | 9.9 | 39.5 | 48.8 |
| 100 | 10 | 63.4 | 95.21 |
| 1000 | 10 | 99.9 | 632 |

Therefore, when we want to apply MSM-1 to a real network (assuming that MSM-1 can topologically arrive at all the peers with the restriction $s = 1$, that the cadence of the source is high enough to provide a new message every $\mu_r$ time units, and that for the number of messages that we have the time delay is lower for MSM-1 than for MSM-2), the algorithm itself should estimate the average number of peers that will receive the message for MSM-1 and if it would not be high enough then it should apply MSM-2, consider the new average number and change if necessary to MSM-3 and so forth.

Nevertheless, the assumption that $P_c$ is equal for all the links has major implications on MSM-1 than on MSM-s for $s \geq 2$. In MSM-1 there are a larger percentage of short end-to-end transmissions than in MSM-s for $s \geq 2$. This means that in MSM-1 there will be in general more transmissions than in MSM-s with a probability of success larger than $P_c$. Thus, the results of Equation 4.13 can be more pessimistic for $s = 1$ than for $s \geq 2$.

# 5. ALGORITHM EVALUATION

In previous chapters we have studied the theoretical aspects of the proposed algorithms. Anyway, we also need to show the algorithms performance, that is, to test their behaviour over a real network. To do this, we have run the algorithms over the Internet network model presented in Section 1.3, slightly extended with the user peers that form the multicast group. Now, we explain the procedure followed to create an overlay network, to execute the algorithms, and to obtain the results.

## 5.1. Backbone graph generator

As said in previous sections, we need first to define a transit backbone. As we have seen in Section 1.3, the Transit-Stub model, that depends on a set of variables of Internet network topology, is good enough for our purposes, and it can be easily used through the Georgia Tech Internet Topology Models (GT-ITM) package [15], a set of applications that are capable of drawing graphs with different topologies according to a variety of probability laws. It also allows to create graphs following the models presented in Section 1.2.

This package works with a generator data file, which contains all the parameters for the backbone model. The output result is a ".gb" file with the graph information (nodes, edges, and costs). Though it is a text file, it is not easy to read, due to its description of nodes and edges. To solve this, we use another tool of the package, which converts the ".gb" file to a ".alt" file which contains "human readable" information. This file will be used by our application to add the user peers (the members of the multicast group), generating the final overlay graph. The GT-ITM package has more tools for the backbone graphs. For example, the ".gb" result file can be converted to use with NS-2 *Network Simulator* [20]; the package can calculate the average of some parameters for a set of backbones; or even we can use another application, the NED *Network Editor* [16], to graphically represent the backbone network.

The result graph topology looks like a real Internet topology, with a set of nodes acting as backbone routers, with high-speed links which interconnect the subnetworks (or stubs). These subnetworks represent the ISP networks, or corporate LANs, which have access networks connected to the backbone network. Finally, we have to add to the ISP networks the user peers (that is, the nodes that will form the multicast group).

## 5.2. MSM Algorithms-Simulator application

### 5.2.1. Application description

To obtain the overlay network, to apply the algorithms over it, and to obtain the results, we use a Java application created for this project. Its name is "MSM

Algorithms Simulator", and it has been implemented with Java and the Eclipse RCP. We chose the Java programming language [17] because, thought it is a bit slower than other languages (for example, C language), it is easy to use (it there exists a lot of libraries with full documentation, and the memory-management is automated), and also has a great portability (since it is an interpreted language, any Java application can be run on any computer if it has a Java Virtual Machine, which is avalaible for almost every existing Operating System).

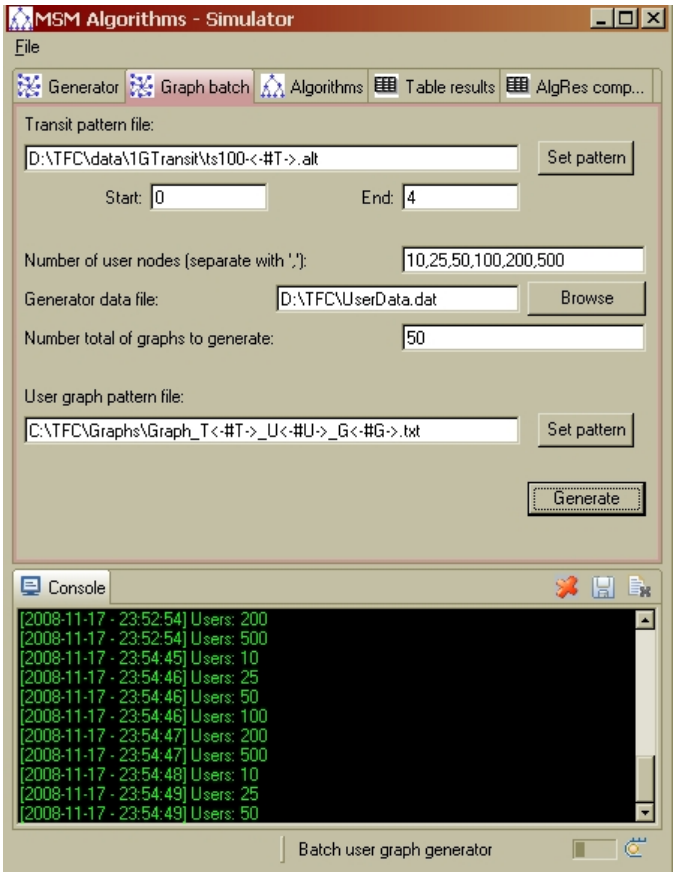



**Fig. 5.1** MSM Algorithms Simulator application layout with the "Graph batch" tab

The use of Eclipse RCP Rich Client Platform [18], [19] allows the development of complex applications with little effort, and with complete GUI interfaces (using different widgets, views and perspectives from a complete UI framework). The platform also allows the developer to add preferences management (including all the "preference-page" related issues), updates management (also with the GUI controls), jobs management (for example, to execute code in separate threads in background), and a lot of other tools like a web-server to use web code in the application (for example, to show the help related to a specific action of the application).

Moreover, with the plugin architecture of the platform, a developer can easily add new features and functions to an existing RCP application. This means that a developer can use the set of connectors from the platform itself, or create new

ones, to add any possible improvement to the application. For example, if a future user wants to create a new algorithm, he only needs to implement it following the definition of its connector; then, without any change in the main application, the new algorithm will be available to use.

The application is used to perform the actions we need to study the algorithms. First, it generates the overlay network over the backbone network acquired in the previous step, then it applies the algorithm on it, and finally it gets the results.

### 5.2.2. Generate overlay network

The first operation of the application is the drawing of the overlay network from an existing ".alt" file that contains the backbone network. The method adds *n* nodes (or peers) which act as "multicast group members" to the backbone network, connecting each of them to one and only one of the stub nodes. The application creates only one single overlay, but it also can be used in batch mode, that is, it also may generate a set of overlays using the same parameters.

This function needs as input data: one backbone graph, the number of peers, and a generator data file. This generator data file contains the characteristics of the user peers that will form the multicast group, like the link bandwidth (either if it is symmetric or not), and the percentage of nodes of each type. This information file sets the parameters for the application, which finally adds each peer in a random way: first, it randomly chooses a node of the backbone network, with an uniform distribution. If this node is a stub node, the new user node is linked to it; if it is a transit node, then another node is choosed. We do this to ensure that all multicast group members are connected to an access network.

Once we have the stub node, we add the link to the new user, with the bandwidth randomly selected from the information file. In this file, we have information about the probabilities for the link bandwidth of the users, and the expected number of users (or the percentage) that will have this link in the resulting overlay graph. The type of the information (speed in kbps or Mbps, symmetric or asymmetric links, absolute number of users or percentage) is set using other parameters of the generator. The application also provides a pattern system to set the name of the files in batch mode. For example, with this pattern, the user can set the name of the overlay graph depending on the backbone graph and the number of users added. This is useful when we want to generate a set of graphs or results to obtain later average results.

### 5.2.3. Algorithm simulation

Next, we must apply our algorithm over the previous generated overlay. The application has a view tab, "Algorithm", which executes one of the available algorithms over a chosen overlay. The algorithm needs the *s* value, and the

identification of the peer that will be used as “root”. Also, a list of algorithms is available, where the user can select the algorithm that he wants to apply. As said before, the RCP platform structure allows to easily add new algorithms to the list.

Like the previous function, the application can execute the algorithms in batch mode, with a set of overlay graphs, and using different algorithms and *s* values.

### 5.2.4. Algorithm results

The next step is to know how the algorithm works over an overlay network. With this aim, the application has an “Algorithm results” view, which presents the relevant parameters of the execution results: the number of connected nodes, the delay for connected nodes, the cadence of any of them, and the routing tree. Also we can know the number of connected nodes that can have congestion difficulties. Recall that a node can suffer from congestion when its cadence time is bigger than the root cadence time, or also when one of the nodes which forms the path from the root to the node has also congestion problems. Again, we can use the application in batch mode to get the average value of the delay or the maximum cadence for a single algorithm and a single *s* value.

### 5.2.5. Results comparation

The last function of the application is used to compare the different algorithms. It generates a table with the relevant information for each execution, with the values averaged over the overlay graphs, and obtaining the minimum *s* value at which the algorithm draws a complete tree, and the *s* value at which the algorithm has a minimum delay.

Some modifications have been made to this function to get tables comparing different parameters, like the number of nodes that can be affected by congestion issues, or the total delay. Anyway, these modifications are not available directly in the RCP application, and only can be executed using the Java application. Anyway, as said before, the addition of this function in the main application is simple and easy.

# 6. RESULTS, CONCLUSIONS AND FURTHER WORK

In this last chapter we define the final overlay networks, we apply over them the algorithms described in Section 3.6, and present the results in terms of time delay and nodes with congestion difficulties. We also compare the different algorithms, we decide which the best is in each case and we explain the reasons of their different performances. We also point which is the best policy in any general case and finally we present a proposal for further work.

## 6.1. Transit backbone parametrization

As seen in Section 5.1, first of all we need a transit backbone as a basis for our overlay network. With this purpose, we have used the GT-ITM tools to build five networks according to the parameters shown in Table 6.1. This represents five different backbone networks based on the Transit-Stub model.

**Table 6.1.** GT-ITM parametrization

| | |
|---|---|
| Transit domains | 1 |
| Av. Nodes (T) / Transit domain | 4 |
| Stub domains / Transit node | 3 |
| Av. Nodes (S) / Stub domain | 8 |
| Edge Prob (between nodes in the same Transit domain) | 0.6 |
| Edge Prob (between nodes in the same Stub domain) | 0.42 |
| Transit-Stub extra edges | 0 |
| Stub-Stub extra edges | 0 |
| Transit-Transit bandwidth | 1 Gbit/s |
| Transit-Transit propagation delay | 100 ms |
| Transit-Stub bandwidth | 100 Mbit/s |
| Transit-Stub propagation delay | 10 ms |
| Stub-Stub bandwidth | 100 Mbit/s |
| Stub-Stub propagation delay | 10 ms |

## 6.2. Data for overlay graphs generation

To obtain a realistic network model as a testbed for the algorithms, in addition to the use of the Transit-Stub model for the backbone, we have also applied the generator application described in Section 5.2. We have consider the user parameters from the results of a survey in February 2006 by the AIMC (*Asociación para la Investigación de Medios de Comunicación*) [21]. Then, we represent the user peers as ADSL users, who are connected to an ISP network. The values used in the generator user data file are shown in Table 6.2.

We also consider that the propagation delay for the links that join a peer with a stub node is 1 ms, since the peer nodes and the access network are usually

very close together. In the table, the bandwidth speeds are symmetric, whereas in a real ADSL they are asymmetric. Nevertheless, the increment of the upload speeds and the use of high-speed connections, like the future use of FTTH and cable connections, make this restriction valid to test the algorithms. With these values, we have generated 10 overlay graphs for each of the 5 backbone networks with 10, 25, 50, 100 and 200 peers respectively (in total, 50 networks for each number of peers), by using our application.

**Table 6.2.** Survey of ADSL users, February 2006

| Base link | Absolute users | % |
|---|---|---|
| I don't know | 3,725 | 7.1 |
| 56 kbps | 4,951 | 9.4 |
| 64 kbps | 445 | 0.8 |
| 128 kbps | 914 | 1.7 |
| 256 kbps | 2,235 | 4.3 |
| 512 kbps | 5,278 | 10.1 |
| 1 Mbps | 21,811 | 41.6 |
| 2 Mbps | 6,342 | 12.1 |
| 4 Mbps | 5,767 | 11.0 |
| 8 Mbps | 765 | 1.5 |
| Don't know / Don't answer | 165 | 0.3 |

## 6.3. Algorithms results and conclusions

For each overlay graph, we have applied the four algorithms described in Section 3.6, with *s* from 1 to *n* if the number of peers is lower than 25, and 30 if the number of peers is higher. Then, using the application, we obtain some values from the execution results: number of connected nodes, number of times that the root sends every message, cadence time for the root (i.e. the time that the root sends one message), maximum cadence time in the whole tree, and the time transmission *t*[0] for one packet. Also, we count the number of nodes whose cadence time is higher than the root, and the total number of nodes that can be affected by congestion (i.e. the nodes with a cadence time higher than the root plus the nodes which follows some of them in the multicast tree).

In addition to the four algorithm variations described in Section 3.6, we also run the original SMM algorithm, which is independent of the *s* value. It always chooses the nearest peer from all the peers that have already the data, with no special constrictions. Thus, the transmission tree is always full connected, but, as it happens with the MSM algorithm, some peers can have higher cadence time than the root, and thus we can have congestion difficulties. Finally, in the example of Table 6.3, we can see the results of the executions with the fourth overlay network for the first backbone for 100 peers, and with *s*=9.

**Table 6.3.** Example of algorithms results

| Algorithm | Cadence | Forced Leaves | Forced All | Dynamic | SMM |
|---|---|---|---|---|---|
| *S* Min / *S* Best | 6 / 6 | 1 / 5 | 1 / 6 | 10 / 10 | – |
| *S* value | 9 | 9 | 9 | 9 | – |
| Connected | 100 | 100 | 100 | 89 | 100 |
| Root sends | 9 | 9 | 9 | 9 | 11 |
| Root cadence | 99 | 99 | 99 | 111 | 324 |
| Max cadence | 269 | 269 | 269 | 111 | 324 |
| Time *t*[0] | 475 | 475 | 475 | 349 | 474 |
| *b0* | 342 | 342 | 342 | 342 | 342 |
| Excess *b0* | 0 | 0 | 0 | 0 | 0 |
| Excess Root | 12 | 12 | 12 | 0 | 0 |
| Affected nodes | 61 | 61 | 61 | 0 | 0 |

For each graph and each algorithm, we seek the minimum *s* value that allows full connectivity (that is, a complete tree) for the *n* users. In addition, we also seek the minimum *s* optimal value, which is the minimum *s* value for a specific algorithm with the lowest maximum cadence time; as seen before, if the number of packets *M* is high enough, the total transmission time to send all the data will be minimum for the lowest cadence time. In case that two or more results have the same maximum cadence time, then we consider the minimum multicast delay *t*[0]. Finally, if we still have two or more executions with the same maximum cadence time and with the same *t*[0], we consider the number of affected nodes, that is, we choose the value of *s* with a minimum number of affected nodes.

Once we have the best value of *s* for all the algorithms, we seek the algorithm that, for each overlay network, works better. In other words: for a specified number of user nodes we count the times that each algorithm is the best for all the graphs. If two or more algorithms can be considered as the best (they have the same maximum cadence time, the same *t*[0] value and the same number of affected nodes), we count all of them.

The results after averaging the values for all the backbone graphs can be seen in Table 6.4. We show the average of the times that each algorithm gives the best result, and the average of the affected nodes (the nodes that can have congestion difficulties) after using this algorithm.

As can be seen, the Forced Leaves algorithm is the best, followed by the Forced All and the Cadence algorithms. Anyway, the number of affected nodes is quite large, especially for the Forced Leaves algorithm (we can see that, for a large number of nodes, the percentage of affected nodes is higher than 50%). This could be a serious issue, since these nodes could be blocked with the consequent loss of data. This number of affected nodes is large enough to try to find an alternative.

**Table 6.4.** Results with affected nodes

| Users | Algorithm | Times | Affected nodes |
|---|---|---|---|
| 10 | Cadence | 5.8 | 1.81 |
| | Forced Leaves | 10.0 | 2.80 |
| | Forced All | 6.8 | 2.04 |
| | Dynamic | 2.4 | 0 |
| | SMM | 1.6 | 0 |
| 25 | Cadence | 5.6 | 4.06 |
| | Forced Leaves | 10.0 | 8.22 |
| | Forced All | 6.0 | 3.75 |
| | Dynamic | 2.0 | 0 |
| | SMM | 0.8 | 0 |
| 50 | Cadence | 3.6 | 16.22 |
| | Forced Leaves | 10.0 | 24.44 |
| | Forced All | 3.6 | 16.22 |
| | Dynamic | 1.4 | 0 |
| | SMM | 0 | 0 |
| 100 | Cadence | 2.8 | 22.05 |
| | Forced Leaves | 10.0 | 64.30 |
| | Forced All | 2.8 | 22.05 |
| | Dynamic | 1.2 | 0 |
| | SMM | 0 | 0 |
| 200 | Cadence | 2.6 | 102.67 |
| | Forced Leaves | 10.0 | 161.22 |
| | Forced All | 2.6 | 102.67 |
| | Dynamic | 1.0 | 0 |
| | SMM | 0 | 0 |

Then, a new approach has been used to decide which algorithm is the best. If previously the main criterion was the maximum cadence time, now we only take into account the results with no affected nodes, and, from here, we use the same previous criteria. With this restriction, we can find that, for some graphs and/or algorithms, no value of *s* satifies this condition, so the final tree has always at least one node whose cadence time is higher than the root one. But, on the other hand, if this value of *s* exists, it means that we can find a tree transmission that avoids any problem caused by congestion. These results are shown in Table 6.5. It is very similar to the previous table, but now the values in “affected nodes” column will be always 0. Again, the “times” value is the average of times that one algorithm is considered “the best”.

Now the Dynamic algorithm is usually the best (nine out of ten times approximately, as it has been designed to avoid any congestion issue). Thus, if we want to get a transmission tree in which no node exceeds the cadence time of the root, we should use the Dynamic algorithm. Anyway, as it is not always the best algorithm, we can generate the transmission tree for more than one algorithm variation, and compare the results. For example, if we want to use

only two algorithms, we can execute the Dynamic and the Forced All over the same overlay network, and use the best tree. Anyway, in case that we can use any algorithm and the processing time is irrelevant (that is, we have a lot of time to draw the tree), we can use all the algorithms and apply the best one.

**Table 6.5.** Results without affected nodes

| Users | Algorithm | Times |
|---|---|---|
| 10 | Cadence | 4.8 |
| | Forced Leaves | 4.8 |
| | Forced All | 4.8 |
| | Dynamic | 9.8 |
| | SMM | 1.4 |
| 25 | Cadence | 5.0 |
| | Forced Leaves | 5.2 |
| | Forced All | 5.2 |
| | Dynamic | 9.2 |
| | SMM | 1.4 |
| 50 | Cadence | 4.6 |
| | Forced Leaves | 4.6 |
| | Forced All | 5.2 |
| | Dynamic | 9.2 |
| | SMM | 2.4 |
| 100 | Cadence | 3.0 |
| | Forced Leaves | 3.0 |
| | Forced All | 3.0 |
| | Dynamic | 9.0 |
| | SMM | 1.2 |
| 200 | Cadence | 2.4 |
| | Forced Leaves | 2.4 |
| | Forced All | 2.4 |
| | Dynamic | 9.0 |
| | SMM | 1.4 |

The use of this approach to avoid congestion issues may result in thicker trees, changing the total delay time $t[0]$ for one packet and also increasing the maximum cadence time. So, we can point the differences between one case and the other. In Table 6.6 we show the average of the maximum cadence time for the best algorithm in each overlay and the total delay $t[0]$, for the two approaches: when the best algorithm minimizes the total time, but it may have congestion issues; and when the best algorithm ensures that any node will have lower cadence time than the root.

Thus, when we apply the algorithms without affected nodes, the maximum cadence time is higher, resulting in a worse transmission time for large number of packets. Anyway, this cadence time is about 50% higher than the cadence with affected nodes, whereas in this last case the number of nodes that may be

affected by congestion is very high (in fact, the percentage increases with the number of user peers). Then, despite the increment of the total time to transmit the complete data, we advice to use the Dynamic algorithm to avoid the loss of messages due to congestion issues.

**Table 6.6.** Time results

| User nodes | Type | Max cadence | *t[0]* | Affected nodes |
|---|---|---|---|---|
| 10 | With affected | 176.94 | 516.08 | 2.80 |
| | Without affected | 226.66 | 480.80 | – |
| 25 | With affected | 208.48 | 541.00 | 8.22 |
| | Without affected | 287.46 | 532.82 | – |
| 50 | With affected | 214.00 | 598.76 | 24.44 |
| | Without affected | 305.94 | 586.10 | – |
| 100 | With affected | 214.00 | 602.74 | 64.30 |
| | Without affected | 323.32 | 597.40 | – |
| 200 | With affected | 214.00 | 612.62 | 161.22 |
| | Without affected | 325.24 | 607.34 | – |

After previous results, now we discuss why some algorithms are better than others. For the case with affected nodes in Table 6.4, the best algorithm (that is, the one which obtains the best solution more frequently) is the Forced Leaves, followed by the Forced All. The reason is that the Forced Leaves only forces to transmit the nodes which have not yet send the packet, whereas Forced All can force any node, which includes upper nodes from the tree (that is, nodes with a higher cadence time, in general). Hence, the maximum cadence time will be usually higher in Force All, and so the total transmission time.

Moreover, the Forced All and the Cadence algorithms obtain similar results. In the Cadence algorithm we need a nimimum *s* to get complete connectivity with all peers, while in any of the Forced algorithms the connectivity is achieved with *s*=1. To get a complete tree with the Cadence algorithm, we may increment *s* to allow a higher time limit and let the peers enough time to send the packet to all of them. This increment of *s* may increase the cadence time for the upper peers of the tree (that is, the nearest peers to the root), and hence the cadence time may also be increased. The use of the Forced All algorithm is similar, with the difference that the increment of *s* is made “automatically” when we need it, but again the upper peers will have more branches, and thus the cadence time will be similar for both algorithms. Finally, the Dynamic algorithm is the most restrictive of all, because a node can never have higher cadence time than the root. Then, to get a complete tree, the *s* value may be higher, as it can be seen in Table 6.7, which results in higher cadence times.

Table 6.7 shows, for all users and algorithms, the average of the *s* values that make the tree full connected, the *s* that gives the best time result (for each of the algorithms), and the *s* at which the algorithm builds a tree where no node exceeds the root cadence. Note that, in this last case, some algorithms may not

fulfill the condition, and so at the side of the average number of *s* we point the number of times that the algorithm fulfills it (over a maximum value of 50, for 10 overlay graph on each of the 5 backbone graphs).

**Table 6.7.** *s* values for the algorithms

| Users | Algorithm | *s* minimum | *s* best | *s* for not affected nodes |
|---|---|---|---|---|
| 10 | Cadence | 3.58 | 3.86 | 3.58 (36) |
| | Forced Leaves | 1.00 | 3.04 | 3.56 (36) |
| | Forced All | 1.00 | 3.34 | 3.56 (36) |
| | Dynamic | 4.12 | 4.12 | 4.12 (50) |
| | SMM | 1.00 | 1.00 | 1.00 (32) |
| 25 | Cadence | 5.38 | 5.62 | 5.71 (28) |
| | Forced Leaves | 1.00 | 4.38 | 5.76 (29) |
| | Forced All | 1.00 | 4.80 | 5.76 (29) |
| | Dynamic | 6.40 | 6.40 | 6.40 (50) |
| | SMM | 1.00 | 1.00 | 1.00 (20) |
| 50 | Cadence | 5.34 | 5.68 | 6.50 (30) |
| | Forced Leaves | 1.00 | 4.16 | 6.10 (30) |
| | Forced All | 1.00 | 5.18 | 6.10 (30) |
| | Dynamic | 6.08 | 6.08 | 6.08 (50) |
| | SMM | 1.00 | 1.00 | 1.00 (23) |
| 100 | Cadence | 5.60 | 5.84 | 5.88 (16) |
| | Forced Leaves | 1.00 | 4.40 | 5.88 (16) |
| | Forced All | 1.00 | 5.70 | 5.63 (16) |
| | Dynamic | 7.32 | 7.32 | 7.32 (50) |
| | SMM | 1.00 | 1.00 | 1.00 (15) |
| 200 | Cadence | 5.82 | 5.86 | 7.69 (16) |
| | Forced Leaves | 1.00 | 4.80 | 7.69 (16) |
| | Forced All | 1.00 | 5.86 | 7.69 (16) |
| | Dynamic | 8.56 | 8.56 | 8.56 (50) |
| | SMM | 1.00 | 1.00 | 1.00 (16) |

According to the results without affected nodes, the best algorithm is Dynamic, as we said before, since it has been defined to avoid congestion issues. After it, the best algorithm is Forced All, followed by Forced Leaves and Cadence (actually, these three algorithms may not be considered in many cases because they do not fulfill the constriction of not having affected nodes, as seen in Table 6.7). Anyway, these three algorithms obtain very similar results, and the *s* values are not very different than the Dynamic ones. Moreover, the number of times that these algorithms draw a complete tree with no nodes affected by congestion is much lower than Dynamic, which always finds complete trees whose nodes have a cadence time lower than the root one. Recall that Dynamic algorithm needs a higher *s* since it is the most restrictive one. After it, the Cadence algorithm needs the second higher *s* because it is the second most restrictive algorithm, whereas Forced All and Forced Leaves guarantee connectivity for *s*=1. Moreover, for "*s* best" we have a little bit higher value for

Forced All than for Forced Leaves. This is because in Forced All we have in general a higher cadence time for the upper peers, which finally results in a worse transmission time than the time for the Forced Leaves algorithm.

As conclusion, we can consider two possible cases to optimize the total transmission time: first, try to do it as fast as possible, using the Forced Leaves algorithm that, on average, is the best one and draws complete trees, but which has the drawback that it may cause congestion.

On the other hand, we can use the other approach: try to find the quickest way to transmit any information, but with the guarantee that we never could have congestion issues. To do this, we use the Dynamic algorithm, which has been the best performing algorithm in a large percentage of cases; and which always fulfills the cadence restriction in such a way that it always avoid congestion.

## 6.4. Further work

We have shown in previous sections that, depending on the constrictions used to avoid congestion, one algorithm may be better than the others. Nevertheless, we must keep in mind that all the algorithms are based on the same main principles, described in Section 3, which were designed to optimize the delay time to send a single packet. Anyway, as it has been seen in previous sections, when we want to send a package set (which will be the usual case in data transmissions) the main objective is to minimize the cadence time. Thus, we could describe another algorithm –also based on SMM– with the objective of minimizing the maximum cadence time of the nodes (again trying to avoid congestion issues) instead of the total delay $t[0]$ for one single packet.

In addition, we could describe new scenarios as a basis for the simulator, using other generator data to model the present ADSL services, or even more sophisticated environments, like a college network or a corporate one.

With respect to the algorithm, we could consider dynamic effects like connection and disconnection of multicast group members, to know how it could affect the whole tree. At this point, some approaches could be used: apply the algorithm again from the beginning when there is a change, or try to connect the new nodes or those which have lost their source to one of the other peers (for example, the root, or one of the leaves of the tree). Thus, the program could be modified to allow, once the original tree has been build, the connection and disconnection of nodes with the aim of preserving full connectivity. At this point, we can also consider the network availability. Thus, in the case that a network link breaks down, we could try to restore the transmission tree as soon as possible with the least loss of data.

Finally, as a direct application of this work, we could apply the proposed algorithms in a real network, and check if theoretical results correspond to their performance in a real network.

# BIBLIOGRAPHY REFERENCES

[1] S. E. Deering and D. R. Cheriton, “Multicast routing in datagram internetworks and extended LANs”, *ACM Trans. Comput. Syst.*, vol. 8, no. 2, pp. 85-110, 1990.

[2] M. H. Ammar, “Why Johnny can’t multicast: lessons about the evolution of the internet” in *NOSSDAV ’03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, New York, NY, USA: ACM Press, pp. 1-1, 2003.

[3] J. H. Saltzer, D. P. Reed and D. D. Clarck, “End-to-end arguments in system design”, *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277-288, 1984.

[4] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole, Jr., “Overcast: Reliable multicasting with an overlay network”, in *USENIX Symposium on Operating Systems Desing and Implementation*, pp. 197-212, 2000.

[5] Y. Chawathe, S. McCanne, and E. A. Brewer, “RMX: Reliable multicast for heterogeneous networks”, in *INFOCOM.* IEEE, pp. 795-804, 2000.

[6] A.-M. K. M. Castro, P. Druschel and A. Rowstron, “SCRIBE: A large-scale and decentralised application-level multicast insfrastructure”, *IEEE Journal on Selected Areas in Communication (JSAC)*, vol. 20, no. 8, October 2002.

[7] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, “Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination”, in *NOSSDAV*, June 2001.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content addressable network”, in *ACM SIGCOMM*, 2001.

[9] A. Bar-Noy, and S. Kipnis, “Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems” in *ACM Symposium on Parallel Algorithms and Architectures,* pp. 13-22, 1992.

[10] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, “How to Model an Internetwork” in *IEEE Infocom*, pp. 594-602, IEEE, San Francisco, 1996.

[11] Gibbons, *Algorithmic Graph Theory,* Cambridge University Press, Cambridge, 1985.

[12] R. J. Wilson, *Introducción a la Teoría de Grafos,* Alianza Universidad, Alianza Editorial, Madrid, 1972.

[13] J. Ozón, *Contribución al coloreado de grafos y las redes pequeño mundo*, Tesis Doctoral, Universitat Politècnica de Catalunya, 2001.

[14] B. Huarte, *Optimización de rutas para el transporte urbano de mercancías,* Projecte Final de Carrera ETSETB, Barcelona, 2004.

[15] GT-ITM, Modeling Topology of Internetworks, "http://www.cc.gatech.edu/computing/Networking/projects/gt-itm/".

[16] NED Network EDitor, "http://www.cs.nyu.edu/pdsg/projects/partitionable-services/ned/ned.htm".

[17] Java Technology, "http://java.sun.com".

[18] Eclipse, http://www.eclipse.org/

[19] Jeff McAffer, Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding and Packaging Java Applications*, Addison-Wesley Professional, 2005. Web site: "http://eclipsercp.org/".

[20] The Network Simulator NS-2, "http://www.isi.edu/nsnam/ns".

[21] AIMC, Asociación para la investigación de medios de comunicación, http://www.aimc.es/.

[22] E. W. Dijkstra, "A note on two problems in connextion with graphs", *Numerische Mathematik,* vol. 1, pp. 269-271, 1959.