Computing
DOI 10.1007/s00607-012-0195-y

Distributed optimization of P2P live streaming overlays

Amir H. Payberah · Jim Dowling · Fatemeh Rahimain · Seif Haridi

Received: 7 November 2011 / Accepted: 5 June 2012

Abstract Peer-to-peer live media streaming over the Internet is becoming increasingly more popular, though it is still a challenging problem. Nodes should receive the stream with respect to intrinsic timing constraints, while the overlay should adapt to the changes in the network and the nodes should be incentivized to contribute their resources. In this work, we meet these contradictory requirements simultaneously, by introducing a distributed market model to build an efficient overlay for live media streaming. Using our market model, we construct two different overlay topologies, tree-based and mesh-based, which are the two dominant approaches to the media distribution. First, we build an approximately minimal height multiple-tree data dissemination overlay, called Sepidar. Next, we extend our model, in GLive, to make it more robust in dynamic networks by replacing the tree structure with a mesh. We show in simulation that the mesh-based overlay outperforms the multiple-tree overlay. We compare the performance of our two systems with the state-of-the-art NewCoolstreaming, and observe that they provide better playback continuity and lower playback latency than that of NewCoolstreaming under a variety of experimental scenarios. Although our distributed market model can be run against a random sample of nodes, we improve its convergence time by executing it against a sample of nodes taken from the Gradient overlay. The evaluations show that the streaming overlays converge faster when our market model works on top of the Gradient overlay.

A. H. Payberah (✉) · J. Dowling · F. Rahimain · S. Haridi
SICS, Isafjordsgatan 22, Box 1263, 164 29 Kista, Sweden
e-mail: amir@sics.se

J. Dowling
e-mail: jdowling@sics.se

F. Rahimain
e-mail: fatemeh@sics.se

S. Haridi
e-mail: seif@sics.se

Keywords P2P live streaming · The Gradient overlay · Distributed algorithms · Market-based algorithms · Auction algorithm

Mathematics Subject Classification 68M14

1 Introduction

Live media streaming over the Internet is getting more popular everyday. The conventional solution for such applications is the client–server model, which allocates servers and network resources to each client request. However, providing a scalable and robust client–server model, such as Youtube, with more than one billion hits per day[1], is very expensive. There are few companies, who can afford to provide such an expensive service at large scale. An alternative solution is to use *IP multicast*, which is an efficient way to multicast a media stream over a network, but it is not used in practice due to its limited support by Internet Service Providers. The approach, used in this work, is *Application Level Multicast* (*ALM*), which uses *overlay networks* to distribute large-scale media streams to a large number of clients (*nodes*). A *peer-to-peer* (*P2P*) overlay is a type of overlay network in which each node simultaneously functions as both a client and a server to the other nodes in a network. In this model, nodes who have all or part of the requested media can forward it to the requesting nodes. Since each node contributes its own resources, the capacity of the whole system grows when the number of nodes increases.

Live media streaming using P2P overlays is a challenging problem. To have a smooth media playback, data blocks should be received with respect to certain timing constraints. Otherwise, either the quality of the playback is reduced or its continuity is disrupted. Moreover, in live streaming, it is expected that at any moment, nodes receive points of the media that are close in time, ideally, to the most recent part of the media delivered by the provider. For example, in a live football match, people do not like to hear their neighbours celebrating a goal, several seconds before they can see the goal happening. Satisfying these timing requirements is more challenging in a dynamic network, where nodes join/leave/fail continuously and concurrently, and the network capacity changes over time. Furthermore, the nodes should be incentivized to contribute and share their resources in a P2P overlay, otherwise, the opportunistic nodes, called *free-riders*, can take advantage of a system without contributing to content distribution.

Many different solutions are already proposed for P2P media streaming, but few of them are able to satisfy all the above mentioned requirements. We believe this is partly because some of these requirements are conflicting. For example, in order to provide a constant high quality stream, nodes should store the media in their buffer for a while before they start to play; which will result in a high playback latency and start up delay.

In this work, we present our P2P live media streaming solution in the form of two systems: *Sepidar* [35] and *GLive* [32]. In Sepidar, we build multiple approximately minimal height overlay trees for content delivery, whereas, in GLive, we build a mesh overlay, such that the average path length between nodes and the media source is

[1] http://www.thetechherald.com/article.php/200942/4604/YouTube-s-daily-hit-rate-more-than-a-billion.

approximately minimum. In these streaming overlays, the nodes with higher available upload bandwidth that can serve relatively more nodes are positioned closer to the media source. This structure reduces the average distance from nodes to the media source; reducing both the probability of a streaming disruptions and playback latency at nodes. Nodes are also incentivized to provide more upload bandwidth, as nodes that contribute more upload bandwidth have relatively higher playback continuity and lower latency than the nodes further to the media source.

We first describe an Integer Linear Programming (ILP) formulation of the topology building problem and provide a centralized solution for it based on the auction algorithm [7], and later we propose a distributed market model to solve the problem at large scale. Our distributed market model differs from the centralized implementations of the auction algorithm, in that we do not rely on a central server with a global knowledge of all participants. In our distributed model, each node, as an auction participant, has only partial information about the system. Nodes continuously exchange their information, in order to acquire more knowledge about other participating nodes in the system.

There are different options for how communication between nodes could be implemented. For example, a naive solution could use flooding, but it is costly in terms of bandwidth consumption, and therefore is not scalable. On the other hand, the communication could be based on random walks or sampling from a random overlay, but we show in the experiments in the Sect. 6 that random sampling has slow convergence time. To enable faster convergence of the streaming overlay, our distributed market model acquires knowledge of the system by sampling nodes using the gossip-generated *Gradient overlay* network [37,38]. The Gradient overlay facilitates the discovery of neighbours with similar upload bandwidth.

The free-riding problem, as one of the problems in P2P streaming systems, is considered in our model. Nodes are not assumed to be cooperative; nodes may execute protocols that attempt to download data blocks without forwarding it to other nodes. We address this problem in Sepidar through parent nodes auditing the behaviour of their child nodes in trees. We also address free-riding in GLive by implementing a scoring mechanism that ranks the nodes. Nodes who upload more of the stream have relatively higher score. In both solutions, nodes with higher rank will receive a relatively improved quality. We do not, however, address the problem of nodes colluding to receive the video stream.

Our contributions in this work include:

- an ILP formalization for modeling a streaming overlay construction and a centralized auction-based solution for it,
- a distributed solution for the presented ILP model to construct streaming overlays, firstly as a multiple-tree overlay, Sepidar, and secondly as a mesh-based overlay, GLive,
- improving the convergence time of the distributed solution by using the Gradient overlay in comparison with a random network,
- two solutions to overcome the free-riding problem in tree-based and mesh-based overlays.

In the next section we describe the related work and position our systems in the field. In Sects. 3 and 4 we explain the problem and present the centralized auction algorithm to solve it, and then in Sect. 5 we go into the details of our systems and explain the distributed solution to build the overlay streaming. In Sect. 6 we present the results of our experiments, and finally we conclude the work in Sect. 7.

2 Related work

There are three fundamental questions in building an overlay for live media streaming: (i) what overlay topology is built for data dissemination? (ii) what algorithm is used for data dissemination? and (iii) how to construct and maintain the streaming overlay and discover the supplying nodes?

Many different overlay topologies have been used for data delivery in P2P media streaming systems. Early data delivery overlays use single-tree structures that the data blocks are pushed over a tree-shaped overlay with the media source as a root of the tree. Climber [31], ZigZag [41], NICE [3], and [16] are examples of such systems. The short latency of data delivery is the main advantage of this approach [49]. Disadvantages, however, include the fragility of the tree structure upon the failure of nodes close to the root and the fact that all the traffic is only forwarded by the interior nodes. Multiple-tree structure is an improvement for single-tree overlays where the first time was proposed in SplitStream [10]. In this model, the stream is split into sub-streams and each tree delivers one sub-stream. Sepidar [35], gradienTv [34], Orchard [27], ChunkySpread [43] and CoopNet [28] are some other solutions in this class.

Although multiple-tree overlays improve the shortcoming of the single-tree structures, they are still vulnerable to the failure of interior nodes. Rejaie et al. have shown in [24] that mesh overlays have consistently better performance than tree-based approaches for scenarios where there is churn and packet loss. The mesh structure is highly resilient to node failures, but it is subject to unpredictable latencies due to the frequent exchange of notifications and requests [49]. GLive [32], Gossip++ [14], NewCoolStreaming [19], Chainsaw [29], and PULSE [36] are the systems that use a mesh-based overlay for data dissemination.

The second fundamental question in building a P2P streaming system is how to distribute data blocks among the nodes. *Pushing* the data is a solution used mostly in tree structures. ZigZag [41] and SplitStream [10], as instances of the single-tree and multiple-tree structures, use push model for data dissemination. However, the push model in mesh-based overlays may generate loads of redundant messages, since nodes may receive the same data block from different neighbours. Therefore, *pulling* data is the dominant data distribution model in mesh-based overlays. In the pull model, the nodes exchange their data blocks availability information and request each required data block explicitly from a neighbour that possesses that data block. Sepidar and GLive use push and pull data distribution model, respectively. There is also a *hybrid* model for data dissemination that combines push and pull mechanisms. Some of the systems that utilize hybrids model are CliqueStream [2], mTreebone [46], NewCoolStreaming [19], Prime [23] and [21].

The third question is how to create an overlay and how nodes discover the other nodes that supply the stream. CoopNet [28] uses a centralized coordinator, GnuStream [17] uses controlled flooding requests, SplitStream [10] and [21] use DHTs, ZigZag

[41] uses a hierarchal model, while NewCoolstreaming [19], DONet/Coolstreaming [51] and PULSE [36] use a gossip-generated random overlay network to search for the nodes. Recently, there has been work on using gossiping to build non-random mesh topologies, where the topology stores implicit information about node characteristics, such as upload bandwidth. In [13], Fortuna et al. attempt to organize nodes with decreasing upload bandwidth at increasing distance from the source. As such, these systems have similarities with how Sepidar and GLive that use the Gradient overlay to structure nodes.

In our systems, we use a market model to optimize partnering nodes for media streaming. Our market model is inspired by auction algorithms. The first widely-used auction algorithm was designed by Bertsekas [7], and has an equivalent representation as a weighted bipartite matching problem [47]. However, our model is an example of a distributed auction algorithm with partial information. Moreover, our model differs from existing work, such as [50] and [30], in that all nodes are decision makers, the set of tasks and resources are homogeneous and auctions are restartable.

The problem of reducing free-riding in P2P systems has been solved by many existing incentive mechanisms and reputation models [25,27,39]. Of particular relevance to Sepidar and GLive is Give-to-Get [26] that uses transitive dependencies to a child's children in order to audit children nodes. In contrast to our systems, Give-to-Get is a video-on-demand protocol based on BitTorrent. Tan and Jarvis also in [39] describe a payment-based approach to solving free-riding for live streaming. Nodes run periodic auctions for their resources and earn points that can be used to access resources. Whereas we incentivize nodes to provide more resources to get better video performance, they incentivize nodes to remain in the system even when not viewing video to acquire an increased number of points. Another related approach to matching nodes for live streaming is based on finding maximal bipartite matchings using a flow algorithm by Li and Mahanti [20]. They transformed the traditional min-cost media flow dissemination problem into an auction problem.

3 Problem description

In this paper, we want to build a P2P overlay for live media streaming and adaptively optimize its topology to minimize the average *playback latency* and improve timely delivery of the stream. Playback latency is the difference between the playback time (*playback point*) at the *media source* and at a node. In this section, we first describe this problem in the tree-based approach, and then present the required modification to apply it for the mesh-based approach.

In our tree-based model, the media stream is split into a number of sub-streams or *stripes*, and each stripe is divided into *blocks* of equal size without any coding. Sub-streams allow more nodes to contribute bandwidth and enable a more robust system, due to redundancy [10]. Every block has a sequence number that indicates its playback order in the stream. A node retrieves the stripes independently, from any other node that can supply them.

We define the number of *download-slots* and *upload-slots* of a node as the number of stripes that a node is able to simultaneously download and forward, respectively.

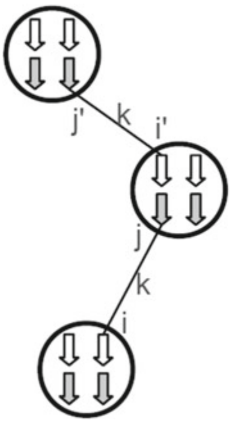


Fig. 1 The connection between download-slots i and i' and upload-slots j and j' for strip k. The *white arrows* are download-slots and the *gray arrows* are the upload-slots

The set of all download-slots and upload-slots in an overlay are also denoted by $\mathcal{D}$ and $\mathcal{U}$, respectively. Similarly, the set of download-slots and upload-slots of a node p is shown by $\mathcal{D}(p)$ and $\mathcal{U}(p)$.

A node is called the *owner* of its slots, and the function $owner(i)$ returns the node that owns the slot i. Any two slots i and j are called *similar*, if they are owned by the same node, i.e., $owner(i) = owner(j)$. For each download-slot i, the set of all download-slots similar to i is called the *similarity class* of i, and denoted by $\mathcal{M}_{\mathcal{D}}(i)$. Likewise, the similarity class of an upload-slot j is the set of upload-slots owned by the owner of j and is shown by $\mathcal{M}_{\mathcal{U}}(j)$.

Without loss of generality, we assume every node owns the same number of download-slots, equal to the number of stripes, and a potentially different number of upload-slots. In order to provide the full media to all the nodes, (i) every download-slot needs to be assigned to an upload-slot, (ii) each upload-slot should be assigned to at most one download-slot, (iii) similar download-slots, i.e., download-slots at the same node, must download distinct stripes, and (iv) nodes should not have loop back connections from their download-slots to their own upload-slots.

This problem can be defined as an *assignment problem* [42]. A connection between a download-slot i and an upload-slot j for a stripe k is shown as a triple (i, j, k), and it is associated with a *cost* c_{ijk} (Fig. 1). The cost can be defined based on different metrics, e.g., the latency to the source, the number of hops to the source or the locality of the nodes, that is a connection between two nodes in the same Autonomous System (AS) has lower cost compare to a connection between two nodes in different ASs. Formally the cost is defined as the following:

$$c_{ijk} = \begin{cases} c_{i'j'k} + d_{ij} & \text{if} \quad owner(j) = owner(i'), i' \in \mathcal{D}, \quad \text{and} \quad j' \in \mathcal{U} \\ 0 & \text{if} \quad owner(j) = \text{source} \\ \infty & \text{if} \quad \text{there is no path from } owner(i) \text{ to the source} \end{cases} \tag{1}$$

where d_{ij} is the added cost in a connection between a download-slot i and an upload-slot j. For example, if the cost is defined as the latency of the node to the source, then

d_{ij} will be the connection latency between the $owner(i)$ and the $owner(j)$, and if the cost is the number of hops to the source, then $d_{ij} = 1$.

With $\mathcal{V}$ being the set of all stripes, we define an *assignment* $\mathcal{S}$ as a set of triples (i, j, k), such that:

1. For all $(i, j, k) \in \mathcal{S}$, $i \in \mathcal{D}$ and $j \in \mathcal{U}$ and $k \in \mathcal{V}$.
2. For each $i \in \mathcal{D}$, there is at most one triple $(i, j, k) \in \mathcal{S}$.
3. For each $j \in \mathcal{U}$, there is at most one triple $(i, j, k) \in \mathcal{S}$.
4. For each $k \in \mathcal{V}$, there is at most one triple $(i, j, k) \in \mathcal{S}$ for all $i \in \mathcal{M}_{\mathcal{D}}(i)$.

The last constraint implies that the download-slots in one similarity class cannot download the same stripe. In other words, each download-slot in a node should download a distinct stripe. Note that with the above definition, it is possible to have cyclic connections among nodes in an assignment.

A *complete assignment* $\mathcal{A}$ is an assignment with the above definition that contains exactly $|\mathcal{D}|$ triples, i.e., all download-slots are assigned. To have a complete assignment, the total number of download-slots should be less than or equal to the total number of upload-slots, i.e., $|\mathcal{D}| \leq |\mathcal{U}|$.

The playback latency of a node depends on the maximum latency of the node in different stripe trees. Therefore, to improve the playback latency we should minimize the latency of a node for all stripes simultaneously. Hence, we use the average distance of a node at all stripe trees as the cost function. Considering a complete assignment $\mathcal{A}$, the cost of a node p is defined as:

$$\mathcal{C}_{\mathcal{A}}(p) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} c_{ijk} \cdot x_{ijk} \tag{2}$$

where $x_{ijk} = 1$ if a download-slot i is assigned to an upload-slot j for a stripe k, and $x_{ijk} = 0$, otherwise. Since putting the nodes with higher upload-slots closer to the source can reduce the average distances of all the nodes to the source [13,34], we bias the cost of each node p by the number of its upload-slots:

$$\mathcal{C}'_{\mathcal{A}}(p) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \tag{3}$$

where $m_i = |\mathcal{U}(owner(i))| = |\mathcal{U}(p)|$ denotes the number of upload-slots that the owner of i has. Then, the average cost of all the nodes in a complete assignment $\mathcal{A}$ is measured as:

$$\begin{aligned}
\mathbb{C}_{\mathcal{A}} &= \frac{1}{|\mathcal{N}|} \sum_{p \in \mathcal{N}} \mathcal{C}'_{\mathcal{A}}(p) \\
&= \frac{1}{|\mathcal{N}| \cdot |\mathcal{V}|} \sum_{p \in \mathcal{N}} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \\
&= \frac{1}{|\mathcal{N}| \cdot |\mathcal{V}|} \sum_{i=1}^{|\mathcal{D}|} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk}
\end{aligned} \tag{4}$$

where $\mathcal{N}$ is the set of all the nodes. Therefore, the problem to be solved turns out to be finding a complete assignment $\mathcal{A}$ among all the possible complete assignments, which minimizes the total cost $\mathbb{C}_{\mathcal{A}}$. Since, the term $\frac{1}{|\mathcal{N}|\cdot|\mathcal{V}|}$ is a constant we ignore it in the optimization process.

Putting all the above constraints together, we formulate the problem in Integer Linear Programming (ILP) framework [5], as the following:

$$\text{minimize} \quad \sum_{i=1}^{|\mathcal{D}|} \sum_{\{j|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \tag{5}$$

subject to

$$\sum_{\{j|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} x_{ijk} = 1, \quad \forall i \in \mathcal{D} \tag{6}$$

$$\sum_{\{i|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} x_{ijk} \leq 1, \quad \forall j \in \mathcal{U} \tag{7}$$

$$\sum_{\{i\in\mathcal{M}_{\mathcal{D}}(i)|(i,j,k)\in\mathcal{A}\}} \sum_{\{j|(i,j,k)\in\mathcal{A}\}} x_{ijk} = 1, \quad \forall k \in \mathcal{V} \tag{8}$$

$$x_{ijk} \in \{0, 1\}, \ \forall i \in \mathcal{D}, \ j \in \mathcal{U}, k \in \mathcal{V} \tag{9}$$

The first constraint requires that every download-slot i is assigned to exactly one upload-slot. The second constraint ensures that each upload-slot is assigned to at most one download-slot. It also stated that if the number of upload-slots are greater than the number of download-slots, some of the upload-slots remain unassigned. The third constraint ensures that the download-slots in a similarity class download distinct stripes.

Our model of the system should consider dynamism, while solving this assignment problem. A good solution, therefore, should assign download-slots to upload-slots as quickly as possible. Centralized solutions to this problem are possible for small system sizes. For example, if all the nodes send the number of their upload-slots to a central server, the server can use an algorithm that solves a linear sum assignments, e.g., the auction algorithm [7], the Hungarian method [18], or more recent high-performance parallel algorithms [42]. For large scale systems, however, a centralized solution is not appropriate, since it can become a bottleneck. In the next section, we briefly sketch a possible solution with the auction algorithm [4,6]. Later in the Sect. 5 we present a distributed model of the auction algorithm that solves this problem at large scale.

4 Streaming overlay construction: the centralized solution

The auction algorithm models a real world auction, where people bid for the objects that brings them the highest profit, and the highest bids win. We use the auction algorithm for $|\mathcal{D}|$ download-slots that compete for being assigned to some upload-slot among the set of $|\mathcal{U}|$ available upload-slots. Like ordinary auctions, the bidders, i.e., the download-slots, progressively increase their bid for the objects, i.e., upload-slots, in a competitive process. However, unlike real world auctions, in our system each download-slot can be assigned to at most one upload-slot.

A matching between a download-slot i and an upload-slot j for stripe k is associated with a *profit* a_{ijk}, and the goal of the auction is to maximize the total profit for all the matchings, which is:

$$\sum_{i=1}^{|\mathcal{D}|} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} a_{ijk} \cdot x_{ijk} \tag{10}$$

where x_{ijk} is defined in Eq. 2, and a_{ijk} is calculated as:

$$a_{ijk} = \frac{m_i}{c_{ijk}} \tag{11}$$

Note that m_i and c_{ijk} are already defined in Sect. 3. Hereafter, we refer to m_i as *money*. Equation 11 simply says that a connection with a lower cost is more desirable, and the more money a download-slot has, the more profit it gets by creating a connection to a lower cost upload-slot. In other words, by maximizing Formula 10, we minimize the cost function in Formula 5.

Download-slots have a certain amount of money, with which they find a matching that maximizes their profit. Each upload-slot j is associated with a *price* p_j. The price of an unassigned upload-slot is zero, and is increased in auction iterations after accepting new *bids* from download-slots (the bidding process will be described later in this section). We define the *net profit* of an upload-slot as its profit minus its current price. A download-slot i measures the net profit, v_{ijk}, of each upload-slot j for a stripe k as the following:

$$\begin{aligned} v_{ijk} &= a_{ijk} - p_j \\ &= \frac{m_i}{c_{ijk}} - p_j \end{aligned} \tag{12}$$

The auction algorithm proceeds in iterations and creates a sequence of assignments (one assignment $\mathcal{S}$ in each iteration), such that the net profit of each connection under the assignment $\mathcal{S}$ is maximized. In each iteration, the algorithm also updates the price of all upload-slots in the assignment $\mathcal{S}$. If all the download-slots of an assignment $\mathcal{S}$ are assigned, the algorithm terminates, otherwise some download-slot i, which is unassigned, finds an upload-slot that offers maximal net profit. Note that at the beginning of each iteration, the net profit of each connection (Eq. 12) under the assignment $\mathcal{S}$ should be maximum.

Each iteration in the auction algorithm contains two phases: a *bidding phase* and an *assignment phase*:

– **Bidding phase**
  In this phase each unassigned download-slot i under the assignment $\mathcal{S}$ finds the upload-slot j^* with highest net profit for a stripe k, where k is not assigned to any other download-slots $i \in \mathcal{M}_{\mathcal{D}}(i)$:

$$v_{ij^*k} = \max_{j \in \mathcal{U}} v_{ijk} \tag{13}$$

The download-slot i also finds the second best upload-slot j' for stripe k, such that j' is not owned by the owner of j^*, i.e., $j' \notin \mathcal{M}_{\mathcal{U}}(j^*)$. The second best net profit, $w_{ij'k}$, equals:

$$w_{ij'k} = \max_{j \notin \mathcal{M}_{\mathcal{U}}(j^*)} v_{ijk} \tag{14}$$

Considering δ_{ij^*k} as the difference between the highest net profit and the second one, i.e., $\delta_{ij^*k} = v_{ij^*k} - w_{ij'k}$, the download-slot i computes the bid, b_{ij^*k}, for the upload-slot j^*:

$$b_{ij^*k} = p_{j^*} + \delta_{ij^*k} \tag{15}$$

In this process at each iteration the download-slot i raises the price of a preferred upload-slot j^* by the bidding increment δ_{ij^*k}.

– **Assignment phase**
The upload-slot j, which received the highest bid from i^*, removes the connection to the download-slot i' (if there was any connection to i' at the beginning of the iteration), and assigns to i^*, i.e., the connection (i^*, j, k) is added to the current assignment $\mathcal{S}$. The upload-slot j also updates its own price to the received bid from the download-slot i^*, i.e., $p_j = b_{i^*jk}$.

Lemma 1 *If* $\delta_{ijk} > 0$*, the auction process will terminate.*

Proof If an upload-slot j receives n bids during n iterations, its price p_j increases by $\sum_1^n \delta_{ijk}^{(n)}$, where $\delta_{ijk}^{(x)}$ is the added price at iteration x. Therefore, over the iterations the upload-slot j becomes more and more "expensive" and consequently its net profit decreases. This implies that an upload-slot can receive bids only for a limited number of iterations, while some other upload-slots still have not received any bids. Hence, after some iterations, $|\mathcal{D}|$ distinct upload slots will receive at least one bid. Bertsekas shows in [5] that an auction algorithm with m bidders, where the set of bidder–object pair is limited, terminates once m distinct objects receive at least one bid. □

However, if $\delta_{ijk} = 0$, it may happen that several download-slots compete for the same set of upload-slots without raising the price, thereby they may stuck in an infinite loop of bidding and assignment phases. To solve this problem, each download-slot that bids for an upload-slot, should rise the price by a small value ϵ by biding $b_{ij^*k} = p_{j^*} + \delta_{ij^*k} + \epsilon$. The details of how ϵ is selected is out of the scope of this paper and can be found in [5].

Although the presented auction algorithm was shown for the multiple-tree approach, we can easily use it to build a mesh overlay. In contrast to the multiple-tree approach, in the mesh-based overlay, we do not split the stream into stripes. The video is divided into a set of $\mathcal{B}$ blocks of equal size without any coding. Every block $b \in \mathcal{B}$ has a sequence number that indicates its playback order in the stream. Nodes can pull each block independently, from any other node that can supply it. Each node has a *partner list*, which is a small subset of nodes in the system. A node can create a bounded number of download connections, equals to its number of download-slots, to partners and accept a bounded number of upload connections, equals to its number of upload-slots, from partners over which blocks are downloaded and uploaded, respectively.

Unlike the tree-based approach that assigns download-slots to upload-slots of nodes for each stripe, here, we need to find the assignments for each block. Biskupski et al. in [8] show that a block disseminated through a mesh overlay follows a tree-based diffusion pattern for each block. Therefore, the objective is to minimize the cost function for every block b, such that a shortest path tree is constructed over the set of available connections for every block. We define the cost of connection c_{ijb} from a download-slot i to an upload-slot j for a block b as the minimum distance, e.g., the number of hops, from the owner of upload-slot j to the media source.

If the set of nodes and the upload bandwidth of all nodes are static, then we can solve the same assignment problem per block. However, P2P systems typically have churn (nodes join and fail) and available bandwidth at nodes changes over time, so as shown in the next section, we have to solve a slightly different assignment problem every time a node join, exits or a node bandwidth changes.

Since the auction algorithm is centralized, it does not scale to many thousands of nodes, as both the computational overhead of solving the assignment problem and communication requirements on the server become excessive, breaking our real-time constraints [42]. In Sect. 5, we present a distributed market model as an approximate solution to this problem.

5 Streaming overlay construction: the distributed solution

In this section, we present the distributed market model to construct the multiple-tree and mesh overlays for media streaming.

5.1 Multiple-tree overlay

Our distributed market model is based on minimizing costs (Eq. 5) through nodes iteratively bidding for upload-slots. We define a node q as the *parent* of a *child* p, if an upload-slot of q is bounded to a download-slot of p. Nodes in this system compete to become children of nodes that are closer to the media source, and parents prefer children nodes who offer to forward the highest number of copies of the stripes. A child node explicitly requests and pulls the first block it requires in a stripe from its parent, and the parent, then, pushes to the child subsequent blocks in the stripe, as long as it remains the child's parent. Children proactively switch parents when they get more net benefit by changing their parents.

We use the following three properties, calculated at each node, to build the multiple-tree overlay:

1. *Money* The total number of upload-slots at a node. A node uses its money to bid for a connection to another node's upload-slot for each stripe.
2. *Price* The minimum money that should be bid when trying to establish a connection to an upload-slot. The price of a node that has an unused upload-slot is zero, otherwise the node's price equals the lowest money of its already connected

children. For example, if node p has three upload-slots and three children with monies 2, 3 and 4, the price of p is 2. Moreover, the price of a node that has a *free-riding* child, a node not contributing in data dissemination, is zero.

3. *Cost* The cost of an upload-slot at a node for a particular stripe is the distance between that node and the media source (root of the tree) for that stripe. Since the media stream consists of several stripes, nodes may have different costs for different stripes. The closer a node is to the media source for a stripe, the more desirable parent it is for that stripe. However, other metrics, such as the nodes' locality, can be taken into account for measuring the cost. For example, if two nodes have the same distance to the source, the cost of choosing any of them by the nodes in the same Autonomous System (AS) is lower than that of the nodes in a different AS. Nodes constantly try to reduce their costs over all their parent connections by competing for connections to the nodes closer to the media source.

Our market model can be best described as an approximate auction algorithm. Similar to the centralized solution in Sect. 4, for each stripe, child nodes place bids for upload-slots at the parent nodes with the highest net profit, e.g., closest nodes to the media source. Note that the money of a node is not used up after bidding and can be reused to bid for other connections. Therefore, if a node can afford a high net profit parent for one stripe, it can also afford other "good" parents for other stripes. However, nodes increase their price by receiving new bids, and the more expensive a node is, the lower net profit it has. Thus, a parent node, which had a high net profit in one iteration, turns out to be a low profit node after receiving a number of bids. Hence, the seeking nodes will try to bid for other nodes with a higher net profit. This implies that in an overlay with $|\mathcal{D}|$ download-slots, if there is no churn in the system, eventually $|\mathcal{D}|$ distinct upload-slots receive at least one bid, and consequently the algorithm terminates by assigning all the download-slots to upload-slots. However, in a dynamic network, where nodes continuously join and leave the system, our algorithm keeps running and optimizes the connections in the overlay.

A parent node sets a price of zero for an upload-slot when at least one of its upload-slots is unassigned. Hence, the first bid for an upload-slot will always win, enabling children to immediately connect to available upload-slots. When all of a parent's upload-slots are assigned, it sets the price for an upload-slot to the money of its child with the lowest number of upload-slots, i.e., the lowest money. If a child with more money than the current price for an upload-slot bids for an upload-slot, it will win the upload-slot and the parent will replace its child with the lowest money with the new child. A child that has lost an upload-slot has to discover new nodes and bid for their upload-slots.

One crucial difference with the auction algorithm is that our market model is decentralized; nodes have only a partial (changing) view of a small number of nodes, called *partners*, in the system with whom they can bid for upload-slots. Moreover, in contrast to the auction algorithm, the price of upload-slots does not always increase - it can be reset to zero, if a child node is detected as a free-rider. A node is free-rider if it is not correctly forwarding all the stripes it promises to supply. As such, it is a "restartable auction", where the auction is restarted because a bidder did not have sufficient funds to complete the transaction.

Algorithm 1 Find candidate parents for stripe k at node p

```
1: procedure findParent ⟨$k$⟩
2:     $candidates = \emptyset$
3:     if $p.stripe_k.parent$ = null then
4:         $p.stripe_k.parent.cost \leftarrow \infty$
5:     end if
6:     for all $n$ in $p.partners$ do
7:         if $n.stripe_k.cost < p.stripe_k.parent.cost$
8:             and $n.price < p.money$
9:             and $n.BM(stripe_k) \geq p.BM(stripe_k)$ then        ▷ $BM$: Buffer Map
10:             $candidates$.add($n$)
11:         end if
12:     end for
13:     return $candidates$
14: end procedure
```

Algorithm 2 Handling the assign request from node p for stripe k at node q

```
1: upon event ⟨ASSIGNREQUEST | $k$⟩ from $p$
2:     if $q.uploadSlots$ has free entries then
3:         assign a free upload slot to $p$
4:         send ⟨ASSIGNACCEPTED | $k$⟩ to $p$
5:     else
6:         if has $freeridingChild$ then        ▷ $q.price = 0$
7:             $lowestMoneyChild \leftarrow freeridingChild$
8:         else        ▷ $q.price =$ the lowest money of the children
9:             $lowestMoneyChild \leftarrow$ the child with the lowest money
10:         end if
11:         if $p.money \leq q.price$ then
12:             send ⟨ASSIGNNOTACCEPTED | $k$⟩ to $p$
13:         else
14:             assign an $uploadSlot$ to $p$
15:             send ⟨RELEASE | $k$⟩ to $lowestMoneyChild$
16:             send ⟨ASSIGNACCEPTED | $k$⟩ to $p$
17:         end if
18:     end if
19: end event
```

To construct the overlay, nodes periodically send their money, cost, price, and *buffer map* to their partners. The buffer map shows the last blocks that a node has in its buffer for different stripes. For each stripe k, a node p periodically checks if it has a node in its partners that has (i) a lower cost than its current parent, (ii) a price less than its money and (iii) blocks ahead of its block in stripe k. As the Algorithm 1 shows, if such a node is found, it is added to a list of candidate parents for stripe k. Next, the node p chooses a node q from the candidates that provides the highest net profit for strip k, i.e., $\frac{p.money}{q.cost_k} - q.price$. If two nodes have the same net profit, it selects the one with higher money.

As the Algorithm 2 shows, if a node q that receives a connection request from node p for stripe k, has a free upload-slot, it accepts the request (lines 2–4), otherwise, if p's money is greater than the price of q, q abandons its child that has the lowest money, and accepts p as a new child. The disconnected node has to find a new parent. If q's price is greater than or equal to p's money, q declines the request (lines 11–17). If q has a free-riding child, q abandons that node as the child with the lowest money (lines 6–7).

Handling free-riders *Free-riders* are nodes that supply less upload bandwidth than claimed. To detect free-riders, we introduce the *free-rider detector* component with *eventual strong completeness* property. By eventual strong completeness property, we mean that, a node that does not have free upload-slots eventually detects all its free-riding children. Nodes identify free-riders through transitive auditing using their children's children (*grandchildren*). To do this, a parent q periodically sends an *audit request*, about its child p, to p's claimed children. Whenever a grandchild receives a message from q, it checks if p is its parent, and has properly forwarded the stripe(s) it has promised to supply. The grandchild, then, sends back either a positive or negative *audit response* to q that shows whether these conditions are satisfied or not. However, this model does not solve the *collusion* problem, if a set of nodes cooperate to cheat.

We now show how the eventual strong completeness property is satisfied for the free-rider detector. Assume a node p claims it has u upload-slots, such that m of them are assigned to other nodes and n of them are free upload-slots, i.e., $u = m + n$. The p's parent, q, periodically sends audit requests to p's m claimed children. Before the next iteration of sending audit requests, q calculates F as the sum of (i) the number of audit responses not received before a timeout, (ii) the number of negative audit responses, and (iii) the n free upload-slots. If F is more than M % of u, p is *suspected* as a freerider. If p becomes suspected in N consecutive iterations, it is *detected* as a free-rider. For example, if N equals 2, a node is detected as a free-rider if it is suspected on two consecutive iterations of the free-rider detector. The higher is the value of N, the more accurate but slower is the detection.

In a converged tree, for nodes not in the two bottom levels (the trees' leaves), we expect that at least M % of their upload-slots are meeting their contracted obligation to correctly supply a stripe over that upload-slot. M is a threshold for free-rider suspicion. For example, if M is 90 %, then node i is suspected as a free-rider, if 10 % or more of its upload-slots are either not connected to child nodes or connected to child nodes but do not supply the stream at the requested rate.

After detecting a node as a free-rider, the parent node q, decreases its own price (q's price) to zero and as a "punishment" considers the free-rider node p as its child with the lowest money. On the next bid from another node, q replaces the free-rider node with the new node. Therefore, if a node claims it has more upload-slots than it actually supplies, it will be detected and punished. In a converged tree, many members of the two bottom levels may have no children, because they are the leaves of the trees, thus, the nodes in these levels are not suspected as free-riders.

5.2 Mesh overlay

To build a mesh overlay, we keep the definition of the price as it is in Sect. 5.1, but we redefine the money and cost as the following [32]:

1. *Money* The total number of blocks uploaded to children during the last 10 s.
2. *Cost* The cost of a node is the average distance of the node to the media source via its shortest path from each of its download-slot. We can also add the locality property to the cost, as mentioned in the tree-based approach.

Like the multiple-tree approach, each node periodically sends its money, cost, price and the buffer map to all its *partners*, which are its neighbours in the mesh. The buffer

Algorithm 3 Handling the parent response message from node q at node p

```
upon event ⟨RECVPARENTMSG | $msg$⟩ from $q$
    if $msg$ is $assignAccepted$ then
        if $p.downloadSlot$ has free entries then
            $p.parents.add(q)$                                  ▷ add $q$ to the parent list
        else
            $wp$ ← the lowest net profit parent                 ▷ the worst parent
            if $(\frac{p.money}{q.cost} - q.price) > (\frac{p.money}{wp.cost} - wp.price)$ then
                $p.parents.remove(wp)$
                $p.parents.add(q)$
                send ⟨REMOVEMEFROMYOURCHILDRENLIST | $p$⟩ to $wp$
            else
                send ⟨REMOVEMEFROMYOURCHILDRENLIST | $p$⟩ to $q$
            end if
        end if
    end if
end event
```

map in the mesh approach shows the list of available blocks in a node buffer. For each of its download-slots, a child node p sends a bid request to those nodes that (i) have a lower cost than the existing parents assigned to download-slots in p, (ii) their price is less than p's money, and (iii) their blocks are ahead of blocks of node p.

A parent node, who receives a bid request, accepts it if (i) it has a free upload-slot, or (ii) it has assigned an upload-slot to another node with a lower amount of money. The pseudo-code is similar to the Algorithm 2, with a small difference that in the mesh approach there is no notion of stripes. If a parent re-assigns a connection to a node with more money, it abandons the old child who must then bid for a new upload connection. The parent behaviour in case of having free-rider child is explained later in this section.

When a child node receives the acceptance message from another node, it assigns one of its download-slots to an upload-slot of the parent. However, since a node may send more connection requests than its number of download-slots, it might receive more acceptance messages than it needs (Algorithm 3). In this case, if the child has a free download-slot, it accepts the parent (lines 3–4), otherwise, it checks all its assigned parents and finds the one with the lowest net profit or the *worst parent*. If the net profit of the connection to the worst parent is lower than the new parent, the child node releases the connection to the worst parent and accepts the new one, otherwise it ignores the received message (lines 7–13).

Handling free-riders Whenever a node assigns a download-slot to an upload-slot of another node, it sends the address of its current children to its parent. It subsequently informs its parents of any changes in its children. Thus, a parent node knows about its children' children, or *grandchildren* for short.

We implement a *scoring* mechanism to detect free-riders, and thus motivate nodes to forward blocks. Each child assigns a score to each of its parents that shows the amount of blocks they have received from their parents in the last 10 s. When a child requests and receives a non-duplicate block from a parent within the last 10 s, it increments the score of that parent. Hence, the more blocks a parent node sends to its children, the higher score it has among its children. We chose 10 s as it is the same as the choking period in BitTorrent [11] and does not unnecessarily punish nodes because of variance in the rate of block forwarding.

Each node periodically sends a *score request* to its grandchildren, and the grandchildren nodes send back a *score response* containing the scores of the original node's children. The node sums up the received scores for each child. Free-rider nodes forward a lower number of blocks, and hence they have lower scores compared to others.

When a node with no free upload-slots receives a connection request, it sorts its children based on their latest scores. If an existing child has a score less than a predefined threshold, s, then that child is identified as a free-rider. The parent node abandons the free-rider nodes and accepts the new node as its child. If there is more than one child with score less than s, then the lowest score is selected. If all the node's children have a score higher than s, then as explained in the previous section, the parent accepts the connection, if the connecting node has more money than the lowest money of its existing children. When the parent accepts such a connection, it then abandons the child with the lowest money. The abandoned child then has to search for and bid for a new connection to a new parent.

Data dissemination Each parent node periodically sends its buffer map and its *load* to all its assigned children. The load shows the ratio of the number of blocks that a node has forwarded to the number of its upload connections.

A child node, uses the information received from its parents to schedule and pull the required blocks in different iteration. We define a *sliding window* that shows the number of blocks that a child node can request in each iteration. If the playback point of a node is t, and the sliding window size is n, the node can request the blocks from t to $t + n$ in each iteration.

One important question in pulling blocks is the order of requests. There are a number of studies [9,52] on block selection policies. The main constraint in data dissemination in live media streaming is that the blocks should be received before their playback time. Therefore, a node should pull the missing block with the closest playback time first, that is, blocks should be pulled in-order. Another potential strategy, as used by BitTorrent [11], is to pull the rarest blocks in the system, as this is known to increase aggregate network throughput [44].

We have designed a download policy that attempts to marry the benefits for playback latency of in-order downloading with the improved network throughput of rarest-block policy. We divide the sliding window into two sets: an *in-order set* and a *rare set*. The first m blocks in the sliding window are the blocks in the in-order set and the rest of the blocks of the sliding window are the rare set blocks. As the names of these sets imply, blocks from the in-order set are requested in order and the least popular block (from among the node's partners) is chosen from the rare set. A node selects a block from the in-order set with probability h % and from the rare set with $(100 - h)$ %, where h is a system parameter. If multiple parents can provide a block, the child node chooses the parent that has the lowest load.

5.3 The Gradient overlay as a market-maker

The problem with a decentralized implementation of the auction algorithm is the communication overhead in nodes discovering the node with the upload-slot of highest net profit. The centralized auction algorithm assumes that the cost of communicating with all nodes is close to zero. In a decentralized system, however, communicating with all nodes requires flooding, which is not scalable. An alternative approach to compute an approximate solution is to find good upload-slots based on random walks or sampling from a random overlay. However, such solutions typically have slow convergence time, as we show in Sect. 6.

It is important that nodes' partial views enable them to find good matching parents quickly. We use the Gradient overlay [37,38,40] to provide nodes with a constantly changing partial view of other nodes that have a similar number of upload-slots. Thus, rather than have nodes explore the whole system for better parent nodes, the Gradient enables nodes to limit exploration to the set of nodes with a similar number of upload-slots.

The Gradient overlay is an overlay network that arranges nodes using a local utility function at each node, such that nodes are ordered in descending *utility values* away from a core of the highest utility nodes [37,38]. The highest utility nodes are found at the center of the Gradient topology, while nodes with decreasing utility values are found at increasing distance from the center. The Gradient is built by both gossiping and sampling from a random overlay network. Each node maintains a set of neighbours called a *similar-view* containing a small number of nodes whose utility values are close to, but slightly higher than, the utility value of the node. Nodes periodically gossip to exchange and update their similar-views.

Node references stored in the similar-view contain the utility value for the neighbours. In our systems, the utility value of a node is calculated using two factors: (i) a node's upload bandwidth, i.e., node's money and (ii) a disjoint set of discrete utility values that we call *market-levels*. A market-level is defined as a range of network upload bandwidths. For example, in Fig. 2, we define 5 example market-levels: mobile broadband (64–127 Kbps) with utility value 1, slow DSL (128–511 Kbps) with utility value 2, DSL (512–1,023 Kbps) with utility value 3, fiber (>1,024 Kbps) with utility value 4, and the media source with utility value 5. A node measures its upload band-

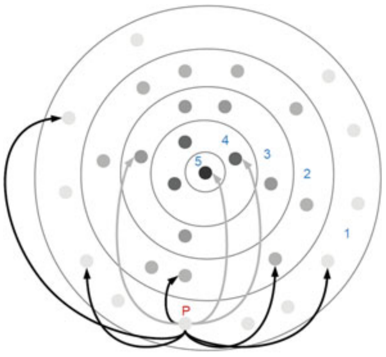


Fig. 2 Different market-levels of a system, and the similar-view and fingers of *p*

Algorithm 4 Updating the similar-view at node p

```
1: procedure UpdateSimilarView ⟨⟩
2:     $p.similarView.updateAge()$
3:     $q \leftarrow$ oldest node from $p.similarView$
4:     $p.similarView.remove(q)$
5:     $pView \leftarrow p.similarView.subset()$
6:     send $pView$ to $q$
7:     recv $qView$ from $q$
8:     for all $n$ in $qView$ do
9:         if $U(n) = U(p)$ or $U(n) = U(p) + 1$ then        ▷ $U(n)$: the utility value of node $n$
10:            if $p.similarView.contains(n)$ then
11:                $p.similarView.updateAge(n)$
12:            else if $p.similarView$ has free entries then
13:                $p.simialrView.add(n)$
14:            else
15:                $m \leftarrow pView.poll()$
16:                $p.similarView.remove(m)$
17:                $p.simialrView.add(n)$
18:            end if
19:        end if
20:    end for
21:    for all $n$ in $p.randomView$ do
22:        if $U(n) = U(p)$ or $U(n) = U(p) + 1$ then
23:            if $p.similarView$ has free entries then
24:                $p.simialrView.add(n)$
25:            else
26:                $m \leftarrow (x \in p.similarView$ such that $U(x) > U(p) + 1)$
27:            end if
28:            if $(m \neq null)$ then
29:                $p.similarView.remove(m)$
30:                $p.simialrView.add(n)$
31:            end if
32:        end if
33:    end for
34: end procedure
```

width (e.g., using a server or trusted neighbour) and calculates its utility value as the market-level that its upload bandwidth falls into. For instance, a node with 256 Kbps upload bandwidth falls into slow DSL market-level, so its utility value is 2. Nodes may also choose to contribute less upload bandwidth than they have available, causing them to join a lower market-level.

A node prefers to fill its similar-view with nodes from the same market-level or one level higher. A feature of this preference function is that low-bandwidth nodes only have connections to one another. However, low bandwidth nodes often do not have enough upload bandwidth to simultaneously deliver all stripes/blocks in a stream. Therefore, in order to enable low bandwidth nodes to utilize the spare upload-slots of higher bandwidth nodes, nodes maintain a *finger list*, where each *finger* points to a node in a higher market-level (if one is available). We illustrate the market-levels and fingers in Fig. 2. Each ring represents a market-level, the black links show the links within the similar-view and the gray links are the fingers to nodes in higher market-levels.

In order for nodes to be able to explore to find new nodes with which to execute our market model, a node constantly updates its neighbours within its market-level. Algorithm 4 is executed periodically by a node p to maintain its similar-view using its *random-view*. The random-view of a node is a random sample of the nodes in the system, which is updated by a peer sampling service, e.g., Cyclon [45], Gozar [33] or Croupier [12].

The algorithm describes how on every round, p increments the age of all the nodes in its similar-view. It removes the oldest node, q, from its similar-view and sends a subset of nodes in its similar-view to q (lines 3–6). Node q responds by sending back a subset of its own similar-view to p. Node p then merges the view received from q with its existing similar-view by iterating through the received list of nodes, and preferentially selecting those nodes in the same market-level as p or at most one level higher. If the similar-view is not full, it adds the node, and if a reference to the node to be merged already exists in p's similar-view, p just refreshes the age of its reference. If the similar-view is full, p replaces one of the nodes it had sent to q with the selected node (lines 8–20). Moreover, p also merges its similar-view with its own local random-view, in the same way described above. Upon merging, when the similar-view is full, p replaces a node whose utility value is higher than p's utility value plus one (lines 21–33).

The fingers to higher market-levels are also updated periodically. Node p goes through its random-view, and for each higher market-level, picks a node from that market-level if there exists such a node in the random-view. If there is not, p keeps the old finger.

Using the Gradient overlay as the market maker, the partners of a node are chosen from the similar-view and finger-list. In other words, in Algorithm 1 (line 6), we should replace $partners$ with $similarView \cup fingers$.

6 Experiments and evaluation

In this section, we compare the performance of Sepidar and GLive with the state-of-the-art NewCoolstreaming [19] under simulation.

6.1 Experimental setup

We have used Kompics [1] to implement Sepidar, GLive and NewCoolstreaming. Kompics is a framework for building P2P protocols and it provides a discrete event simulator for simulating them using different bandwidth, latency and churn models. We have implemented NewCoolstreaming based on the system descriptions from [48].

In our experimental setup, we set the streaming rate to 512 Kbps, which is divided into blocks of 16 Kb. Nodes start playing the media after buffering it for 15 s, which compares favorably to the 60 s of buffering used by state-of-the-art (proprietary) SopCast [22]. The size of similar-view in Sepidar and GLive and the partner list in NewCoolstreaming is 15 nodes. We assume all the nodes have enough download bandwidth to receive the stream with a full rate, and also all the nodes have the same number of download-slots, which is set to 8. To model upload bandwidth, we assume that each upload-slot has available bandwidth of 64 Kbps and that the number of upload-slots for nodes is set to $2i$, where i is picked uniformly at random from the range 1 to 10. This means that nodes have upload bandwidth between 128 Kbps and 1.25 Mbps. As the average upload bandwidth of 640 Kbps is not much higher than the streaming rate of 512 Kbps, nodes have to find good matches as parents in order for good streaming performance. The media source is a single node with 40 upload-slots, providing five times the upload bandwidth of the stream rate. This setting is based on SopCast's requirement that the media source has at least five times the upload capacity of the stream rate [22].

In our simulations, we assume 11 market-levels, such that the nodes with the same number of upload-slots are located at the same market-level. For example, nodes with two upload-slots (128 Kbps) are the members of the first market-level, nodes with four upload-slots (256 Kbps) are located in the second market-level, and the media source with 40 upload-slots (2.5 Mbps) is the only member of the 11th market-level. Latencies between nodes are modeled using a latency map based on the King data-set [15]. We use the hop count in our experiments to measure the cost function.

In the mesh-based solution, we assume the size of sliding window for downloading is 32 blocks, such that the first 16 blocks are considered as the in-order set and the next 16 blocks are the blocks in the rare set. A block is chosen for download from the in-order set with 90 % probability, and from the rare set with 10 % probability. In Sepidar, we set $N = 2$ and $M = 50$ % for the free-rider detector component, and in GLive, we set the threshold of the score, s, to zero.

In this experiment, we measure *playback continuity* and *playback latency*, which combined together reflect the QoS experienced by the overlay nodes:

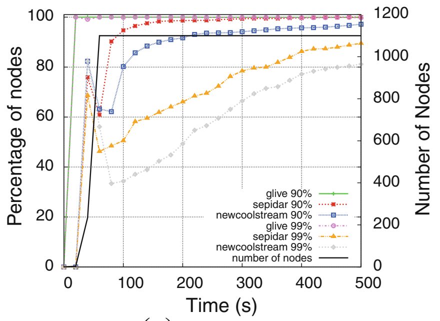
1. *Playback continuity* the percentage of blocks that a node received before their playback time. We consider two metrics related to playback continuity: where nodes have a playback continuity of (i) greater than 90 % and (ii) greater than 99 %;
2. *Playback latency* the difference in seconds between the playback point (the playback time) of a node and the playback point at the media source.

6.2 Sepidar vs. GLive vs. NewCoolstreaming

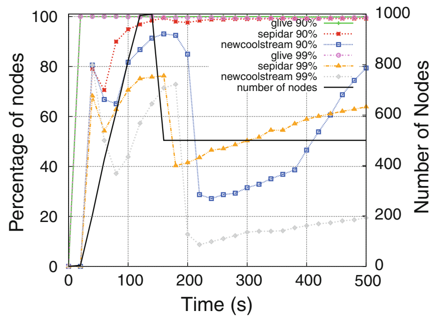
In this section, we compare the playback continuity and playback latency of Sepidar, GLive and NewCoolstreaming in the following scenarios:

1. *Flash crowd* First, 100 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 ms. Then, 1,000 nodes join following the same distribution with a shortened average inter-arrival time of 10 ms;
2. *Catastrophic failure* 1,000 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 ms. Then, 500 existing nodes fail following a Poisson distribution with an average inter-arrival time 10 ms;
3. *Churn* 500 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 ms, and then till the end of the simulations nodes join and fail continuously following the same distribution with an average inter-arrival time of 1,000 ms.

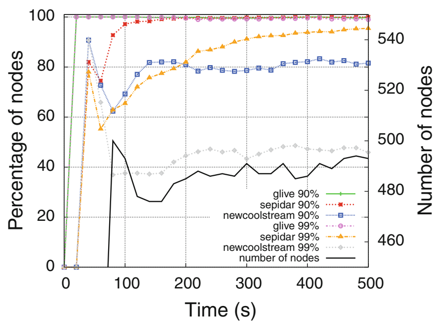
Figure 3 shows the percentage of the nodes that have playback continuity of at least 90 and 99 %. We see that all the nodes in GLive receive at least 90 % of all the blocks very quickly in all scenarios, while it takes more time in Sepidar. That is because in Sepidar, at the beginning, nodes spend time constructing the trees, while



(a) Flash crowd.



(b) Catastrophic failure.

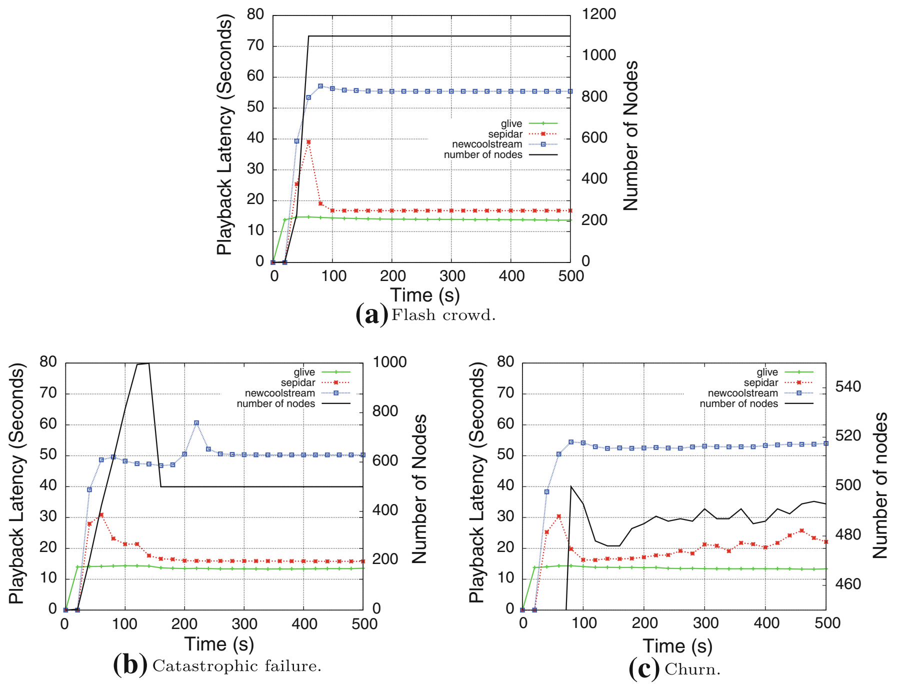


(c) Churn.

Fig. 3 Playback continuity of the systems in different scenarios

in GLive the nodes pull blocks quickly as soon as at least one of their download-slots is assigned. As we see in Fig. 3, both GLive and Sepidar outperform NewCoolstreaming in playback continuity for the whole duration of the experiment in all scenarios. GLive and Sepidar use the Gradient overlay for node discovery. The Gradient overlay arranges nodes based on their number upload bandwidth capacity, and so the neighbours of a node are those with the same upload bandwidth capacity, or slightly higher. This helps the high capacity nodes to quickly discover the media source. In contrast, NewCoolstreaming uses a random overlay, and it takes more time for nodes to find appropriate parents. The result is a higher number of changes in parent connections, causing lower playback continuity in NewCoolstreaming compared to GLive and Sepidar.

As we see in Fig. 3, the difference between GLive and Sepidar increases, when we measured the percentage of the nodes that receive 99 % of the blocks in time. Again, the tree structure used in Sepidar causes this difference. Although, Sepidar has a multiple-tree structure, which is resilient to the failures, it has a lower playback continuity than GLive when nodes crash. In a multiple-tree structure, a node typically receives the blocks of each stripe independently, but if a parent providing a stripe fails, then it loses the blocks from that stripe, while the node is trying to find a new parent for that stripe. However, this problem does not apply to the mesh overlay, because the nodes pull the blocks independently of each other. Therefore, if a node loses one of its parents, it can pull the required blocks from other parents.



(a) Flash crowd.

(b) Catastrophic failure.

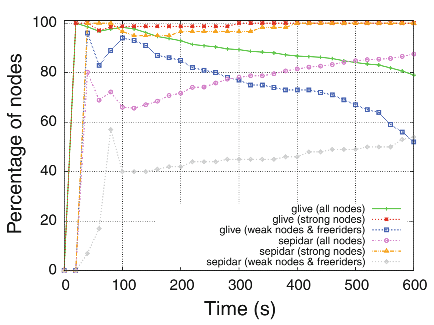
(c) Churn.

Fig. 4 Playback latency of the systems in different scenarios

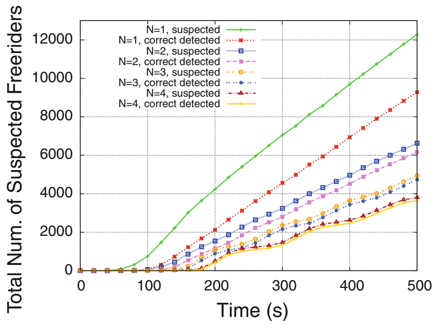
Figure 4 shows the playback latency of the systems in different scenarios. As we can see, GLive keeps its playback latency relatively constant, close to 15 s, which is the initial buffering time. The playback latency of Sepidar also converges to 15 s, but it takes longer to converge than GLive. The reason for this delay is, again, the time needed to construct the trees. The playback latency of GLive and Sepidar, are both less than NewCoolstreaming. In NewCoolstreaming, the higher playback latency is a result of nodes only reactively changing parents when their playback latency is greater than a predefined threshold.

6.3 Free-rider detector settings

Here, we compare the playback continuity of GLive and Sepidar in the *free-rider scenario*. In this scenario, 1,000 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 ms, such that 30 % of the nodes are free-riders, and the total number of upload-slots in the system is less than the total number of download-slots, i.e., $|U| < |D|$. The free-riders can be found in any market-level. Figure 5a shows the percentage of the nodes that receive 99 % of the blocks before their playback time. It shows this value for all the nodes in the system, including the *strong nodes* (top 10 % of upload bandwidth nodes), and the free-riders and the *weak nodes* (the bottom 10 % of upload bandwidth nodes).



(a) Playback continuity in the free-rider scenario.



(b) Sepidar freerider detection settings.

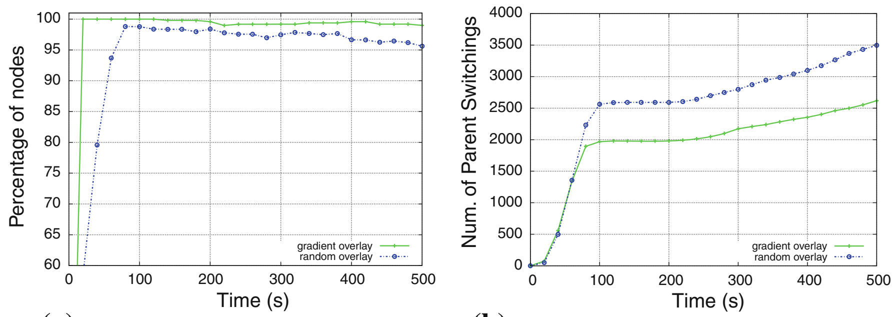
Fig. 5 The systems behaviour in the existence of freeriders

Figure 5a shows that all the strong nodes in both systems receive all the blocks in time, however, GLive converges faster than Sepidar. In GLive, we are using the scoring mechanism to find the nodes who contribute less bandwidth than they claim when bidding for connections, while Sepidar uses a free-rider detector module that identifies nodes that do not meet their contractual requirement to forward the stream to their child nodes. In GLive, at the beginning, a high percentage of weak nodes and free-riders receive all the blocks in time, which shows that free-riders have not been detected yet. That is because nodes need time to update and validate the scores of their parents, and, thus, identify free-riders. Meanwhile, the free-riders use the resources of the system. However, after enough time has passed and the nodes' scores have been updated, the free-riders are detected. Thus, after about 100 s the percentage of the free-riders who have a high playback continuity decreases.

As Fig. 5a shows, after about 600 s from the beginning of the experiment, in both GLive and Sepidar the free-riders and weak nodes receive roughly the same quality of stream, that is, they have the same percentage of playback continuity. As the playback continuity of the weak nodes and free-riders keeps decreasing in GLive, we can also see that the playback continuity decreases for all nodes in GLive. After 500 s playback continuity even decreases below Sepidar.

Importantly, as we can see in Fig. 5a, the existing free-riders in the system have a very low effect on the playback continuity of the strong nodes in Sepidar and GLive. Strong nodes have consistently higher playback continuity than weak nodes and free-riders. This is due to the fact that weak nodes have a lower amount of money compared to strong nodes, which makes them take longer to find good parents. Also, the punishment of free-riders negatively affects their playback continuity. As such, nodes are strongly incentivized to contribute more upload bandwidth through receiving improved relative performance.

In Fig. 5b, we compare the total number of detected free-riders with the number of nodes that are correctly detected as a free-rider in Sepidar for different settings. As we see, with smaller N, the fraction of nodes that are correctly detected as free-riders decreases. There is a trade-off between accuracy and the speed of the detection. Smaller N gives us faster detection but with less accuracy. In this experiment we assume $M = 50\ \%$.



(a) 99% of playback continuity of GLive. **(b)** Total number of parent switches in Sepidar.

Fig. 6 The systems behaviour in the Gradient overlay and random overlay

6.4 Comparing the Gradient with random neighbour selection

In this experiment, we compare the convergence speed of our market model for the Gradient overlay and random overlays. We use the churn scenario in this experiment, as this is the most typical environment for P2P streaming systems on the Internet. Our market model is run using (i) samples taken from the Gradient overlay, where the sampled nodes have similar number of upload-slots, and (ii) samples taken from a random network, where the sampled nodes have random number of upload-slots.

As nodes in the Gradient overlay receive bids from a set of nodes with almost the same money, the difference between received bids is less than the expected difference for the random network. Figure 6a shows that in GLive in the case of using the Gradient overlay, more nodes can quickly receive high playback continuity. As such, the Gradient overlay can be said to be a more efficient market maker for our distributed market model than a random overlay. Figure 6b shows the CDF of number of parent switches in Sepidar for both overlays against time, and we can see that in the Gradient overlay, the system has a substantially lower number of parent switches.

7 Conclusion

In this work we focused on designing and implementing a distributed market model to construct P2P live streaming overlays, in form of tree-based and mesh-based systems, i.e., Sepidar and GLive. Within our streaming systems, we have proposed a distributed market model to construct a content distribution overlay, such that (i) nodes with increasing upload bandwidth are located closer to the media source, and (ii) nodes with similar upload bandwidth become neighbours.

In our model, each node has a number upload-slots and download-slots, and to be able to distribute data blocks to all the nodes, the download-slots of nodes should be assigned to other nodes' upload-slots. We model this problem as an assignment problem. We present a centralized solutions for this problem based on the auction algorithm. However, the centralized solutions are not feasible in large and dynamic networks with real-time constraints, thus, we propose a decentralized implementation of the auction algorithm. We show in the simulation that the decentralized model based on sampling from a random overlay has a slow convergence time. Therefore, we address the problem by using the gossip-generated Gradient overlay to provide nodes with a partial view of other nodes that have a similar upload bandwidth or slightly higher.

We evaluate Sepidar and GLive in simulation, and compare their performance with the state-of-the-art NewCoolstreaming. We show that our solutions provide better playback continuity and lower playback latency than that of NewCoolstreaming in different scenarios. In addition, we compare Sepidar with GLive to highlight the differences of the multiple-tree and the mesh overlays. We observe that the mesh-based overlay outperforms the multiple-tree overlay in all the scenarios. Moreover, we compare the convergence time of our systems when the node samples are given by the Gradient overlay rather than a random network. The experiment results show that the overlays converge faster when our market model works on top of the Gradient overlay. Finally, we evaluate GLive and Sepidar performance in different free-rider settings, and examine the effectiveness of our mechanism for addressing the free-riding problem.

References

1. Arad C, Dowling J, Haridi S (2009) Developing, simulating, and deploying peer-to-peer systems using the kompics component model. In: Proceedings of the 4th ICST international conference on communication system software and middleware (COMSWARE'09), pp 1–9
2. Asaduzzaman S, Qiao Y, Bochmann G (2008) CliqueStream: an efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay. In: Proceedings of the 8th IEEE international conference on peer-to-peer computing (P2P'08), pp 269–278
3. Banerjee S, Bhattacharjee B, Kommareddy C (2002) Scalable application layer multicast. In: Proceedings of the ACM conference on applications, Technologies, architectures, and protocols for computer communication (SIGCOMM'02), pp 205–217
4. Bertsekas D (1992) Auction algorithms for network flow problems: a tutorial introduction. Comput Optim Appl 1:7–66
5. Bertsekas D (1998) Network optimization: continuous and discrete models. Athena Scientific, Belmont
6. Bertsekas DP, Castanon DA (1989) The auction algorithm for the transportation problem. Ann Oper Res 20(1):67–96
7. Bertsekas DP (1988) The auction algorithm: a distributed relaxation method for the assignment problem. Ann Oper Res 14(1):105–123
8. Biskupski B, Schiely M, Felber P, Meier R (2008) Tree-based analysis of mesh overlays for peer-to-peer streaming. In: Proceedings of the 8th IFIP international conference on distributed applications and interoperable systems (DAIS'08), pp 126–139
9. Carlsson N, Eager DL (2007) Peer-assisted on-demand streaming of stored media using bittorrent-like protocols. In: Proceedings of the 6th IFIP international conference on ad hoc and sensor networks, wireless networks, next generation internet (NETWORKING'07), pp 570–581
10. Castro M, Druschel P, Kermarrec AM, Nandi A, Rowstron A, Singh A (2003) Splitstream: high-bandwidth multicast in cooperative environments. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), pp 298–313
11. Cohen B (2003) Incentives build robustness in bittorrent. In: Proceeding of the 1st workshop on economics of peer-to-peer systems (P2PEcon'03), pp 68–72
12. Dowling J, Payberah AH (2012) Shuffling with a croupier: Nat-aware peer-sampling. In: Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS'12)
13. Fortuna R, Leonardi E, Mellia M, Meo M, Traverso S (2010) QoE in pull based P2P-TV systems: overlay topology design tradeoffs. In: Proceedings of the 10th IEEE international conference on peer-to-peer computing (P2P'10), pp 256–265

14. Frey D, Guerraoui R, Kermarrec A, Monod M (2010) Boosting gossip for live streaming. In: Proceedings of the 10th IEEE international conference on peer-to-peer computing (P2P'10), pp 296–305
15. Gummadi K, Saroiu S, Gribble S (2002) King: Estimating latency between arbitrary internet end hosts. In: Proceedings of the 2nd ACM workshop on Internet measurment (SIGCOMM'02), pp 5–18
16. Jarvis SA, Tan G, Spooner DP, Nudd GR (2006) Constructing reliable and efficient overlays for p2p live media streaming. J Simul Process Model 7(2):54–62
17. Jiang X, Dong Y, Xu D, Bhargava B (2003) Gnustream: a p2p media streaming system prototype. In: Proceedings of the IEEE international conference on multimedia and expo (ICME'03), pp 325–328
18. Kuhn HW (1955) The Hungarian method for the assignment problem. Nav Res Logist Q 2(1–2):83–97
19. Li B, Qu Y, Keung Y, Xie S, Lin C, Liu J, Zhang X (2008) Inside the new coolstreaming: principles, measurements and performance implications. In: Proceedings of the 27th IEEE conference on computer communications (INFOCOM'08), pp 1031–1039
20. Li Z, Mahanti A (2006) A progressive flow auction approach for low-cost on-demand p2p media streaming. In: Proceedings of the 3rd ICST international conference on quality of service in heterogeneous wired/wireless networks (QShine'06)
21. Locher T, Meier R, Schmid S, Wattenhofer R (2007) Push-to-pull peer-to-peer live streaming. In: Proceedings of DISC 2007; 21st international symposium on distributed computing, pp 388–402
22. Lu Y, Fallica B, Kuipers F, Kooij R, Mieghem PV (2009) Assessing the quality of experience of sopcast. J Internet Protoc Technol 4(1):11–23
23. Magharei N, Rejaie R (2007) Prime: Peer-to-peer receiver-driven mesh-based streaming. In: Proceedings of the 26th IEEE Conference On Computer Communications (INFOCOM'07), pp 1415–1423
24. Magharei N, Rejaie R, Guo Y (2007) Mesh or multiple-tree: a comparative study of live p2p streaming approaches. In: Proceedings of the 26th IEEE conference on computer communications (INFOCOM'07), pp 1424–1432
25. Meulpolder M, Pouwelse JA, Epema DHJ, Sips HJ (2009) Bartercast: a practical approach to prevent lazy freeriding in p2p networks. In: Proceedings of the 23rd IEEE international symposium on parallel and distributed processing (IPDPS'09), pp 1–8
26. Mol J, Pouwelse J, Meulpolder M, Epema D, Sips H (2008) Give-to-get: Free-riding-resilient video-on-demand in p2p systems. In: Proceedings of the 15th SPIE/ACM Multimedia Computing and Networking (MMCN'08)
27. Mol JJD, Epema DHJ, Sips HJ (2006) The orchard algorithm: P2p multicasting without free-riding. In: Proceedings of the 6th IEEE international conference on peer-to-peer computing (P2P'06), pp 275–282
28. Padmanabhan VN, Wang HJ, Chou PA, Sripanidkulchai K (2002) Distributing streaming media content using cooperative networking. In: Proceedings of the 12th ACM international workshop on network and operating systems support for digital audio and video (NOSSDAV'02), pp 177–186
29. Pai V, Kumar K, Tamilmani K, Sambamurthy V, Mohr AE, Mohr EE (2005) Chainsaw: eliminating trees from overlay multicast. In: Proceedings of the 4th international workshop on peer-to-peer systems (IPTPS'05), pp 127–140
30. Park C, An W, Pattipati KR, Kleinman DL (2010) Distributed auction algorithms for the assignment problem with partial information. In: Proceedings of the 15th international command and control research and technology symposium (ICCRTS'10)
31. Park K, Pack S, Kwon T (2008) Climber: an incentive-based resilient peer-to-peer system for live streaming services. In: Proceedings of the 7th international Workshop on peer-to-peer systems (IPTPS'08), p 10
32. Payberah AH, Dowling J, Haridi S (2011) Glive: the gradient overlay as a market maker for mesh-based p2p live streaming. In: Proceedings of the 10th IEEE international symposium on parallel and distributed computing (ISPDC'11), pp 153–162
33. Payberah AH, Dowling J, Haridi S (2011) Gozar: NAT-friendly peer sampling with one-hop distributed nat traversal. In: Proceedings of the 11th IFIP international conference on Distributed applications and interoperable systems (DAIS'11), pp 1–14
34. Payberah AH, Dowling J, Rahimian F, Haridi S (2010) gradientv: Market-based p2p live media streaming on the gradient overlay. In: Proceedings of the 10th IFIP international conference on distributed applications and interoperable systems (DAIS'10), pp 212–225
35. Payberah AH, Dowling J, Rahimian F, Haridi S (2010) Sepidar: incentivized market-based p2p live-streaming on the gradient overlay network. In: Proceedings of the IEEE international symposium on multimedia (ISM'10), pp 1–8

36. Pianese F, Keller J, Biersack EW (2006) Pulse, a flexible p2p live streaming system. In: Proceedings of the 25th IEEE conference on computer communications (INFOCOM'06), pp 1–6
37. Sacha J, Biskupski B, Dahlem D, Cunningham R, Meier R, Dowling J, Haahr M (2010) Decentralising a service-oriented architecture. Peer-to-Peer Netw Appl 3(4):323–350
38. Sacha J, Dowling J, Cunningham R, Meier R (2006) Discovery of stable peers in a self-organising peer-to-peer gradient topology. In: Proceedings of the 6th IFIP international conference distributed applications and interoperable systems (DAIS'06), pp 70–83
39. Tan G, Jarvis SA (2008) A payment-based incentive and service differentiation scheme for peer-to-peer streaming broadcast. IEEE Trans Parallel Distrib Syst 19(7):940–953
40. Terelius H, Shi G, Dowling J, Payberah AH, Gattami A, Johansson KH (2011) Converging an overlay network to a gradient topology. In: Proceedings of the 50th IEEE conference on decision and control (CDC'11)
41. Tran DA, Hua KA, Do TT (2003) Zigzag: an efficient peer-to-peer scheme for media streaming. In: Proceedings of the 22nd IEEE conference on computer communications (INFOCOM'03), pp 1283–1292
42. Vasconcelos CN, Rosenhahn B (2009) Bipartite graph matching computation on GPU. In: Proceedings of the 7th international conference on energy minimization methods in computer vision and pattern recognition (EMMCVPR'09), pp 42–55
43. Venkataraman V, Yoshida K, Francis P (2006) Chunkyspread: heterogeneous unstructured tree-based peer-to-peer multicast. In: Proceedings of the 14th IEEE international conference on network protocols (ICNP'06), pp 2–11
44. Vlavianos A, Iliofotou M, Faloutsos M (2006) Bitos: enhancing bittorrent for supporting streaming applications. In: Proceedings of the 25th IEEE conference on computer communications (INFOCOM'06), pp 1–6
45. Voulgaris S, Gavidia D, van Steen M (2005) CYCLON: inexpensive membership management for unstructured P2P overlays. J Netw Syst Manag 13(2):197–217
46. Wang F, Xiong Y, Liu J (2007) mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast. In: Proceedings of the 27th IEEE international conference on distributed computing systems (ICDCS'07), p 49
47. West DB (2000) Introduction to graph theory, 2nd edn. Prentice Hall, Upper Saddle River, NJ
48. Xie S, Li B, Keung GY, Zhang X (2007) Coolstreaming: design, theory and practice. IEEE Trans Multimed 9(8):1661–1671
49. Yiu WPK, Jin X, Chan SHG (2007) Challenges and approaches in large-scale p2p media streaming. IEEE Multimed 14(2):50–59
50. Zavlanos MM, Spesivtsev L, Pappas GJ (2008) A distributed auction algorithm for the assignment problem. In: Proceedings of the 47th IEEE conference on decision and control (CDC'08), pp 1212–1217
51. Zhang X, Liu J, Li B, Shing Peter Yum T (2005) Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In: Proceedings of the 24th IEEE conference on computer communications (INFOCOM'05), pp 2102–2111
52. Zhao BQ, Lui JCS, Chiu DM (2009) Exploring the optimal chunk selection policy for data-driven p2p streaming systems. In: Proceedings of the 9th IEEE international conference on peer-to-peer computing (P2P'09), pp 271–280