Hyphen: A Hybrid Protocol for Generic Overlay Construction in P2P Environments *

Mouna Allani
Imperial College London
mallani@doc.ic.ac.uk

Benoît Garbinato
University of Lausanne
benoit.garbinato@unil.ch

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

ABSTRACT

Overlay networks form the core part of peer-to-peer (P2P) applications such as application-level multicast, content distribution and media streaming. To ease development, middleware solutions and toolkit libraries have been proposed in the past to help with the implementation of overlay networks. Existing solutions, however, are either too generic by only providing low-level communication abstractions, requiring developers to implement algorithms for overlay networks from scratch, or too restrictive by only supporting a particular overlay topology with fixed properties. In this paper, we argue that it is possible to find a middle ground between these two extremes.

We describe Hyphen, a middleware for overlay construction and maintenance that supports a range of overlay topologies with custom properties, and show how it can replace topology construction for a variety of application-level multicast systems. Unlike previous efforts, Hyphen can construct and maintain a range of overlay topologies such as trees and forests with specific optimisation goals such as low latency or high bandwidth. By using a gossip-based mechanism to define topologies implicitly, Hyphen can scale to many peers and achieve low construction overhead. Our experimental evaluation with Bullet and Splitstream, two P2P streaming systems, shows that Hyphen can construct a bandwidth-optimised tree for Bullet that achieves a higher streaming rate than the original Bullet implementation, and that it can construct a more reliable forest for Splitstream by taking individual peer reliability into account.

1. INTRODUCTION

In many domains, such as video streaming, file sharing, and content distribution, peer-to-peer (P2P) applications have emerged as a scalable and cost effective alternative to traditional client/server systems. P2P applications are typically implemented on top of *overlay networks*, which act as communication substrate [27]. For example, P2P applications for application-level multicast (ALM) [5] use an overlay network to disseminate messages along a virtual topology. They organise peers into a given logical overlay topology and route disseminated messages through this topology to achieve the desired multicast service.

The implementation of overlay networks as part of P2P applications can be challenging and complex. In addition to basic support in terms of peer discovery, peer membership maintenance and topology construction, the overlay network must handle peer churn and network failures. The constructed overlay network must have a specific virtual topology that is required by the P2P application and provides desired properties in terms of network bandwidth, latency or reliability.

To help with the complexity of overlay networks, libraries that handle different aspects of overlay network implementation such as peer communication or peer membership have been proposed in the past [3, 1]. While these approaches somewhat reduce programming burden, they do not address the question of overlay construction. Existing middleware solutions for overlay constructions are either too restrictive by only supporting specific topologies [25] or too flexible by forcing developers to implement their own algorithms for topology construction and maintenance [26, 29, 30, 24].

However, typically overlay networks share common parts that can be factored out into a middleware layer to simplify the development of overlay-based P2P applications. For instance, tree-like overlay networks (tree, multi-trees and forest) are widely used as the basis for routing in multicast communication. Thus, the complex task of defining and maintaining a tree overlay, for example, is duplicated unnecessarily across a wide range of P2P applications. A middleware layer can avoid duplicate implementation effort for the construction and maintenance of common overlay topologies that different P2P applications share.

A natural question is how to define the boundary between the middleware and the P2P application in order to provide the application with the flexibility to implement specific algorithms. In other words, we need to decide which overlay topologies the middleware should support in order to help with the complexity of overlay network construction without imposing unnecessary restrictions on the application logic of the P2P application. We address this issue by considering the supported *overlay topologies* and associated *optimisation goals*.

Overlay topologies. Different P2P applications rely on

*This research is funded by the Swiss National Science Foundation, in the context of Hyphen project (PBLAP2-132796) at Imperial College London and the University of Lausanne.

SAC'13 March 18–22, 2013, Coimbra, Portugal

different topologies. For instance, SplitStream [9] defines a forest of trees overlay, optimising the available bandwidth use on top of Pastry, which provides a scalable routing overlay. In Bullet [21], a covering random tree is defined for the multicast service on top of which a mesh is then dynamically created by adding links between peers to recover missing messages.

By investigating a number of overlays, we can see that they have a *generic* part that is common across them. A tree overlay is for instance at the heart of several P2P applications. In addition, an overlay also has a *specific* part. For example, while the random covering tree in Bullet is *generic*, the mesh is the *specific* part of the overlay network. These two parts of the Bullet's overlay are separated. The goal of our middleware is to support the implementation of the *generic* parts of overlay networks.

Optimisation goals. An overlay network is constructed with given network properties such as latency and bandwidth that are determined by an *optimisation goal*. A classification of existing overlays [5] revealed that there are three main optimisation goals found across many overlay networks in P2P applications: (1) minimising latency, (2) maximising bandwidth and (3) maximising reliability. Based on this classification, we argue that a flexible middleware solution should be able to consider any of these optimisation goals.

By providing overlays with various optimisation goals, the middleware can not only satisfy the overlay requirements of a P2P application but also further improve the quality of these overlay networks. For example, in the case of Bullet, the middleware can provide a covering bandwidth-optimised tree instead of the default random Bullet tree, offering better throughput. Similarly, SplitStream can leverage a reliable forest topology, which enhances its reliability and minimises the number of times SplitStream must handle churn events.

The Hyphen middleware. As a solution, we propose Hyphen, a flexible middleware for constructing a variety of tree-like overlay networks with specific properties and constraints. When using Hyphen, existing overlay-based P2P applications are not only simplified but can also improve the quality and performance of their communication. Hyphen supports overlays with various optimisation goals, such as minimising latency, maximising reliability or maximising bandwidth. These optimisation goals help existing P2P applications achieve better performance. In addition, overlay networks constructed with Hyphen adapt to changes in their environment.

To ensure that Hyphen's topology construction can scale to a large number of peers, it uses a gossip-style communication model. Gossip protocols distribute the load among all peers in a system. However, traditional random gossip approaches may cause an excessive message overhead because their intrinsic redundancy results in more network traffic. To achieve both low message and computation overheads, Hyphen makes the gossip communication deterministic in order to define and maintain an overlay network implicitly, which we refer to as a *hybrid* approach. Periodic gossip messages are exchanged to construct and maintain the overlay with low computational and message overheads. Each peer selects a specific set of peers to gossip with based on peer properties such as reliability, available bandwidth, etc. The selected peer sets are chosen so that the links among peers form the desired overlay. Several overlay construction techniques already rely on such a hybrid approach to define overlays [12, 32, 23, 7] but they focus on a specific topology and a specific optimisation goal. In this paper, we extend the hybrid approach to build a variety of overlays with various optimisation goals. The main contributions of this paper are listed hereafter.

1. *Flexible overlay construction.* Hyphen supports tree-like overlays that can be used as the basis of various ALM applications. These overlays optimize different properties such as bandwidth, reliability or latency.

2. *Hybrid overlay construction.* To achieve scalability, Hyphen defines and maintains overlays implicitly with low computational and message overheads by exploiting a gossip-style communication.

3. *Experimental evaluation.* Using simulations, we show that when Bullet uses a bandwidth-optimised tree provided by Hyphen, it achieves a higher streaming rate than when using its default random tree. Our results also show that Hyphen can enhance the reliability of SplitStream by providing a forest overlay that takes individual peers' reliability into account.

Roadmap. In the following, Section 2 discusses related work, while Section 3 states our assumptions. In Section 4, we describe the architecture of Hyphen and Section 5 details our algorithms for overlay construction and maintenance. Finally, the evaluation of Hyphen is presented in Section 6 and Section 7 concludes the paper.

2. BACKGROUND

In this section, we place our work in the context of related contributions. First, we describe some middleware solutions supporting overlay-based applications. Then, we describe some popular ALM solutions that could benefit from Hyphen . Finally, we compare Hyphen to existing solutions for overlay construction that use the *hybrid* approach.

Middleware for P2P applications. In seeking of high quality, e.g., low latency, high bandwidth, overlay networks become more complex and hence building them is more challenging. To facilitate construction and deployment of overlay networks, middleware solutions have been proposed, which provide useful abstractions such as communication primitives, group membership maintenance and network metrics tracking [1, 26, 24, 30, 29].

Some middleware solutions offer specification tools and languages that developers can use to define overlay topologies. In Macedon [30], an overlay network is specified by describing its system states, local node states and events using finite state machines. In P2 [26], a declarative language called *Overlog* is proposed to define rules that realise a particular overlay topology. While more concise than Macedon, P2 rules can become complex for non-trivial overlay networks.

More fundamentally, developers using such approaches still need to define algorithms for overlay construction and maintenance using the specification rules provided by the middleware. This means that the overlay definition is considered as part of the overlay-based application development due to the heterogeneity of existing overlays. In contrast, we argue that middleware can provide a higher-level abstraction for overlay networks. A middleware that supports a common set of overlay topologies with a set of performance

properties can simplify the implementation of a range of P2P applications. For example, while specific P2P applications for application-level multicast have their own specific overlay networks, many of these overlays share a set of common parts.

Overlay networks. To understand which parts of an overlay network can be considered generic, we studied a number of different overlay networks. From this study, it clearly appears that *tree-like topologies* are widely used, typically single trees and multi-trees.

Single tree. Tree topologies are widely used because their acyclic property greatly simplifies routing and avoids data redundancy, and thus saves peer resources and reduces produced network traffic. However, tree overlays are sensitive to peer churn: when a non-leaf peer leaves or fails, partitioning occurs, which then interrupts a multicast service. Tree overlays are used either as the main routing overlay [18, 23, 10, 22, 6, 4] or as part of more complex, layered or composed overlay [9, 21, 34]. Existing tree overlays are often constructed based on one or more of three optimisation goals: minimising latency[23, 22], maximising bandwidth [18, 6] and maximising reliability [10, 4].

Multi-tree. A popular example of *multi-trees* is a forest of tree overlays. This topology was first proposed by SplitStream [9] in order to maximise the use of available bandwidth. It represents a special type of multi-tree topology with one source but where each peer is internal in only one tree and is a leaf in the other trees. Based on this condition, the trees of the forest structure are *disjoint*. In [9], the stream is divided into multiple sub-streams named *stripes*. Each stripe is routed through a dedicated disjoint tree of the forest. Forest overlays have been adopted by several recent projects [12, 20] to improve other aspects of overlay-based multicast. Hyphen supports forest-based P2P applications with a forest overlay. In addition, Hyphen is able to deliver a forest overlay with different optimisation goals in order to enhance the quality of this topology. For example, Hyphen can enhance the reliability of SplitStream [9] by providing a more reliable forest overlay.

Mesh. Many overlay-based P2P applications define several layered (or composed) overlays in order to adapt multiple network properties and/or to facilitate routing. In these applications, a mesh overlay is often used in one layer. The mesh layer and its optimisation goals differ from one P2P application to another. Some mesh-based applications first build a mesh overlay and then derive a tree on top of that mesh [11, 28]. Other P2P applications first construct a tree overlay on top of which they dynamically create a mesh by adding links between peers in disjoint sub-trees [21, 13]. A popular example of the latter category is Bullet [21], where a mesh is created on-demand, during a multicast over a tree, in order to recover missing messages. The mesh links consist of new peering relationships created between peers that hold disjoint data. Bullet can function on top of any tree overlay. Originally, a random covering tree was adopted. In this paper, we show that the performance of Bullet can be improved using Hyphen by providing it with a bandwidth-optimised tree.

Hybrid Overlay construction. To overcome the complexity imposed by traditional overlay construction, many recent researches [23, 12, 8] proposed hybrid approaches that combine gossip- and overlay-based strategies. A hybrid protocol diffuses content messages, adapting to node and network constraints as overlay-based protocols, while not imposing an overhead through overlay construction. To do so, a hybrid protocol uses a gossip-style mechanism with deterministic behaviour in the gossip decision. Contrary to traditional random gossip, the subset of selected peers to gossip with is chosen by taking peer properties into account, such as a desired overlay topology. The union of links between selected gossip peers thus *implicitly* defines a specific overlay topology. Examples of systems following a hybrid approach are *Plumtree* [23], which constructs a minimum latency tree, [7], which defines a maximum reliability tree and *Thicket* [12], which defines a forest of minimum latency trees. In contrast to these efforts, Hyphen supports different overlay optimisations and is not restricted to a single topology but supports different tree-like overlays. *T-Man* [19] also supports multiple topologies, such as trees and rings, but constructed overlays do not consider underlying peer properties such as reliability.

3. MODEL

We consider a P2P network composed of N peers that communicate by message passing. More formally, we model the overlay topology as a connected graph $G = (\Pi, \Lambda)$, where $\Pi = \{p_1, p_2, \ldots, p_n\}$ is a set of N peers and $\Lambda = \{l_1, l_2, \ldots\} \subseteq \Pi \times \Pi$ is a set of bidirectional communication links.

Reliability. We assume that peers can crash or leave and links can suffer omission faults. Both process crash and link message loss probabilities are modelled as a *failure configuration* $C = (P_1, P_2, ..., P_n, L_1, L_2, ..., L_{|\Lambda|})$, where P_i is the probability that process p_i crashes during a computation step, and L_j is the probability that link l_j loses a message during a communication step. As explained in Section 4, this system view can be approximated by each process using, for instance, the results presented in [14].

Scalability. To ensure scalability, we assume that a peer p_i knows only its direct neighbours, denoted as N_i. At each peer p_i, the set of direct neighbours N_i represents the peers with which p_i has a peering relationship. The establishment and management of peering relationships is part of a *peering membership mechanism*, which is the protocol executed by peers to join the P2P network and maintain a number of peering neighbours. We assume that knowing about an environment component (link or process) includes knowing all its properties. That is, if a process p_i knows a neighbour n_k then, p_i knows the n_k churn probability, noted P_k. It, also, knows l_k, the link connecting p_i to n_k, its message loss probability L_k and the bandwidth capacity of l_k.

4. THE Hyphen MIDDLEWARE

Hyphen is a flexible middleware that handles overlay construction and maintenance for existing ALM systems. Figure 1 illustrates how Hyphen supports ALM systems (layer 4). Hyphen can be used with existing ALM systems because it supports a range of overlay topologies: single tree, multi-trees and forest. By factoring out overlay construction, Hyphen simplifies the implementation of ALM systems and improves their quality by defining an overlay following different optimisation focus. For this, Hyphen takes the properties of the underlying network peers (layer 0) into account to construct the required overlay topology (layer 2). This process leverages a peer membership mechanism, which associates

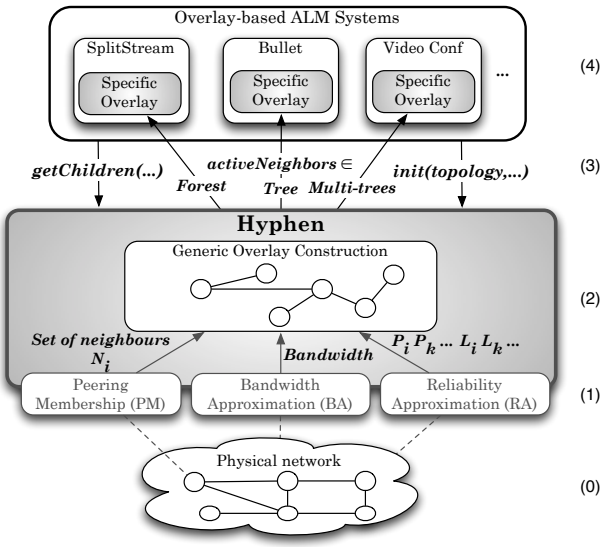


Figure 1: Overview of the Hyphen architecture.

each peer with a set of neighbours and notifies a peer of their arrival or departure. It also leverages a set of approximation mechanisms providing each peer with an up-to-date estimation of the properties within its neighbourhood. These approximation mechanisms continuously refresh the delivered estimations of peer properties. With these estimations, the overlay maintenance process can adapt to changes in the physical network. In what follows, we describe the functionality of the layers illustrated in Figure 1 in bottom up order.

Main building blocks. Hereafter, we describe the main building blocks of Hyphen, as illustrated in Figure 1.

Peering Membership (PM). In a P2P system, a joining peer executes a peering membership algorithm in order to establish a peering relationship with a set of the existing peers. Hyphen uses the peering membership protocol defined in [28], which builds peering relationships in an adaptive manner based on available bandwidth at each peer. Specifically, the degree of each peer (i.e., the number of direct neighbours) depends on the available bandwidth of the peer. Based on that, we assume that each peer has sufficient bandwidth to gossip with all its direct neighbours. To keep a peer's view up-to-date, each peer is notified when a change occurs in the set of its direct neighbours.

Bandwidth Approximation (BA). In order to construct an overlay optimising bandwidth, we assume that each peer receives an up-to-date estimation of the available bandwidth at each direct neighbour. That is, each peer p_i knowns the available bandwidth at any link connecting p_i to a direct neighbour. This approximation could be ensured by one of the several approximation techniques proposed in the literature [16, 17, 15].

Reliability Approximation (RA). In order to maximise the overlay reliability, we assume that each peer p_i has an up-to-date knowledge about its churn probability, noted P_i, the neighbour p_k churn probability, noted P_k, and the message loss probability $L_{i,k}$ of the link $l_{i,k}$ connecting p_i and p_k. We assume this knowledge is provided by an underlying layer. Several techniques could be used to approximate reliability. In [14] for instance, the reliability of peers and links is expressed probabilistically and approximated using Bayesian networks. In IRP [33] and ROST [31] a peer reliability is expressed in terms probability and defined based on the age of the peer. Long-lived peers are considered more reliable.

Generic overlay construction. Hyphen follows a hybrid approach to create and maintain an overlay network implicitly by operating a gossip protocol. That is, Hyphen operates as any pure gossip protocol, in the sense that, in order to build and maintain an overlay a subset of participating peers periodically gossip a set of *control messages* denoted *cmsg*. A distinct periodic gossip of *control messages* is performed for each single tree that is part of the targeted tree-like overlay: single tree, forest or multi-tree. However our gossip is deterministic in that each peer maintains a subset of its neighbors, named *activeNeighbors*, that periodically gossip our control messages *cmsg*. The selection of neighbors to be added to the *activeNeighbors* set ensures that the closure of links among those peers form the *overlay* required by the in-top ALM solution.

More precisely, the gossip of a control message *cmsg* starts by having the source of the tree send *cmsg* to all its neighbours in its *activeNeighbours* set. This implies that some nodes will receive a set of duplicates for the same control message *cmsg*. Based on the quality information brought by these duplicates, some paths are pruned in the overlay so that only paths with high quality are kept. In Hyphen, a peer p_i prunes paths with undesirable properties, from which it received *cmsg*, by removing direct neighbours that provided messages on those paths from its *activeNeighbors* set. Peer p_i also sends a *prune* message to neighbours that sent undesirable *cmsg* messages, so they do not forward subsequent control messages to p_i.

Hyphen APIs. Hyphen constructs and maintain an overlay topology to support an ALM system. When a peer in the ALM system wants to send a message (e.g., a stream packet), it asks Hyphen for the subset of neighbours to which the message should be propagated. This subset is obtained by calling the getChildren function. The union of links between that peer and this subset of its neighbours is guaranteed to be part of the overlay requested by the ALM system. Hence, the full topology is not known to any single peer but it is defined implicitly by the union of gossip peers sets. So Hyphen never exposes the complete overlay topology to the ALM system.

5. THE Hyphen ALGORITHM

In this section, we describe the Hyphen algorithm in detail. Various topologies can be constructed using Hyphen, by passing it the desired overlay topology as a parameter, as suggested in Figure 1. In addition, an overlay can be built with different optimisation goals. For this purpose, we define a *generic* function computing the quality of a path. The quality is *parametric* and can reflect either the bandwidth, the latency or the reliability. Hereafter, we give a description of our *generic* function to compute the quality of a path.

5.1 Parametric Quality Optimisation

As already mentioned, Hyphen aims at supporting different requirements: high reliability, high bandwidth or low latency. Generically, we refer to these different requirements

as Quality Q and we aim at maximising Q. Hereafter, we list the set of parameters related to the quality at a peer p_i.

- $Q_{n,i}$: Quality of the branch connecting p_n to p_i.
- Q_i: Quality of the path from the source to p_i.
- $Q_i^{p_n}$: Quality of the path from the source to p_i via p_n.

Generic quality computation. To compute the quality of a path, we define a function update *Upd* shown in Equation (1). The quality of a path is computed based on the quality of each branch of the path. The *Upd* function returns the quality of a path when adding a new branch to it. For this, it takes as parameters the initial quality of the path Q_n and the quality of the branch $Q_{n,i}$ to be added. The details of the function are described below.

$$Upd(Q_n, Q_{n,i}) = Q_i^{p_n} \quad (1)$$

Specific quality computation. Quality in Hyphen is a generic concept that can reflect different optimisation requirements. Next we describe how the quality is computed depending on targeted optimisations. For this, we define three versions of function *Upd*, denoted Upd_B, Upd_L and Upd_R,which compute the bandwidth, the latency and the reliability of a path from the source to a peer p_i, respectively.

- $Upd_B(Q_n, Q_{n,i}) = min(Q_n, Q_{n,i})$, with $Q_{n,i}$ the bandwidth $l_{n,i}$
- $Upd_L(Q_{n,i}) = \frac{1}{Q_{n,i}+1}$, with $Q_{n,i}$ the time at which p_i received *cmsg* from p_n
- $Upd_R(Q_n, Q_{n,i}) = Q_n \times Q_{n,i}$, with $Q_{n,i} = [(1 - P_n) \times (1 - L_{n,i}) \times (1 - P_i)]$

The bandwidth of a path is computed as the bandwidth of its bottleneck link, which is the link with the lowest bandwidth capacity. To compute the bandwidth of a path when adding a new branch, function *Upd* simply retains the minimum of the bandwidth capacities. The computation of the latency of a path relies only on the time. For this, the function *Upd* considers the reception time of a message through a path: the lower the reception time, the better the latency of the path. To compute the reliability of a path when adding new branch from p_n to p_i, function *Upd* takes into account the reliability the current path Q_n, the churn probability P_n, the churn probability of the target neighbour P_i, and the message loss probability of the link $l_{n,i}$, $L_{n,i}$. The resulting reliability $Q_i^{p_n}$ expresses the probability that a message sent from the source reaches peer p_i.

5.2 Hybrid Overlay Construction

Overlay construction. We now detail the overlay construction algorithm at the heart of the Hyphen middleware. For this, we show an example of the operation of this algorithm in Figure 2. As explained in the previous section, the implicit definition of the overlay relies on the diffusion of control messages *cmsg*. Each control message *cmsg* holds a set of information necessary for the overlay definition such as the quality Q of the path serving this message. To define a tree-like overlay, we associate each tree of the overlay with a *flow* of control messages *cmsg*. Each peer maintains a set of flows corresponding to the trees on which that peer is included. In a single tree overlay, only one flow is defined at each peer. In a multi-tree overlay, the number of flows corresponds to the number of trees. In a forest overlay, the number of flows corresponds to the number of stripes of the forest. At each peer, a *flow* data structure includes the set *activeNeighbours*, with which the current peer gossips multicast messages and control messages, and the set *backupNeighbours* used to ensure connectivity of the tree. In addition, each *flow* data structure indicates the quality Q perceived so far in the path serving the current flow messages. A flow is created first at the source of the corresponding tree. We initialise the quality of a flow at the source peer to 1. Note that, when a flow is initialised, all neighbours are in the *backupNeighbours* set. Before disseminating control messages *cmsg*, a subset of these neighbours moves to the *activeNeighbours*.

The initialisation of the *activeNeighbours* set is different depending on the target topology overlay, i.e., tree, multi-tree or forest. In a single tree or in a multi-tree, this set is initialised with all direct neighbours N_i. This means that our periodic gossip starts initially as a flooding where the first control message is sent to all neighbours. For the forest topology, however, the disjointedness condition should be verified. That is, each peer can be an internal peer in only one flow. At a peer p_i, this means that p_i can have only one flow where the size of its *activeNeighbours* set exceeds 1. In that flow, the *activeNeighbours* set is also initialised with the set of direct neighbours N_i. For any other flow f', this set contains only one element—the neighbour that provides p_i with f' control messages.

In the scenario shown in Figure 2, a flow tree rooted at peer p_1 is to be constructed. Here p_1 can be either the root of a single tree overlay or the root of a tree included in a multi-tree overlay or the root of a stripe part of a forest overlay. A flow f tree construction begins when the flow source peer p_1 sends a control message *cmsg* to all peers in its *activeNeighbours* set (see Figure 2 (a)). The *activeNeighbours* set of p_1 includes all its direct neighbours (i.e., $\{p_2, p_3\}$). To each of its neighbours, the peer p_1 sends *cmsg*, with the quality Q perceived traversed by *cmsg*. At the source the quality is set to 1.

Upon receiving *cmsg*, both p_2 and p_3 update the quality Q of the path traversed by *cmsg*, respectively, up to p_2 and p_3 by calling the update function *Upd*. Then, p_3 forwards the message to all peers in its *activeNeighbours* set except peer p_1, i.e., it forwards *cmsg* only to p_4 (see Figure 2 (b)). In addition, peer p_3 sends the adjusted Q of the path traversed by *cmsg* to reach p_3. Similarly, peer p_2 forwards the message to p_4 (see Figure 2 (c)). When receiving a duplicate of *cmsg* from p_2, p_4 compares Q of the path traversed by *cmsg* from p_2 and Q of the path along which *cmsg* was previously sent (via p_3). In this scenario, we assume that the path via p_2 has a higher quality. To select the path via p_2, p_4 sends a *prune* message to the previous sender of *cmsg*: p_3 (see Figure 2 (d)). When receiving this *prune* message, p_3 removes p_4 from its *activeNeighbours* set. The resulting tree defined by the *activeNeighbours* sets is shown in Figure 2 (e).

Overlay healing. In any overlay-based solution, a peer's churn may cause the partition of the overlay. Thus, to ensure the service availability, a *reactive* strategy to reconnect the overlay is required. Here, we briefly describe a simple healing mechanism inspired by [23, 12].

First, the detection of a partition in tree of flow f relies on the periodic exchange of a summary of the received *cmsg* messages between each peer and its neighbours in the *backupNeighbours* set of f. When a peer p_i receives a summary, it verifies if all messages exist in its f's *receivedMsgs* set. For each missed message *missed*, p_i waits for a timeout

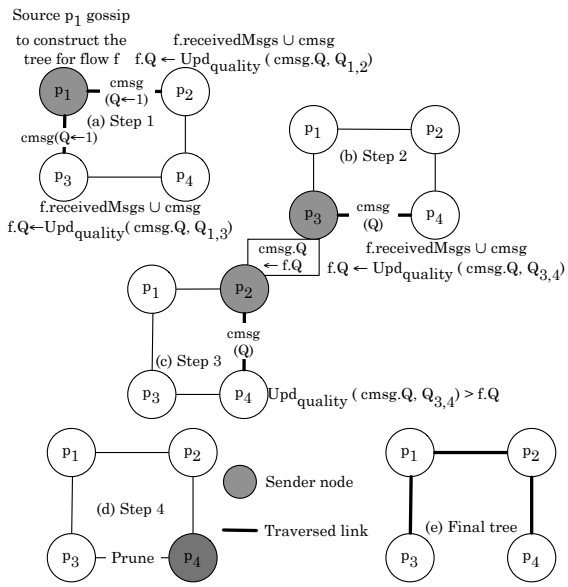


Figure 2: Example of the execution

to receive it through its current flow sender. If the message *missed* has not been received, p_i assumes that it either has never been included in the tree of f or has been disconnected from that tree. To incorporate or heal the tree, p_i sends a *Graft* message to all *backupNeighbours* that announced the message *missed* to p_i in their summaries. When receiving a *Graft* message from p_i, the *backupNeighbour* moves p_i from its *backupNeighbours* set to its *activeNeighbours* set and sends the control message *missed* to p_i.

Since the *missed* message can be announced by several neighbours, several *backupNeighbours* can reconnect p_i. In this case, our mechanism selects the path with the best quality being optimised by the current overlay. Note that to preserve the required topology, e.g., a forest, only peers that are willing to accept new children send their summary of the received *cmsg* messages periodically. In other words, a peer p_i will not announce received messages of flow f if p_i is unable to serve neighbours with f's messages.

6. EVALUATION OF Hyphen

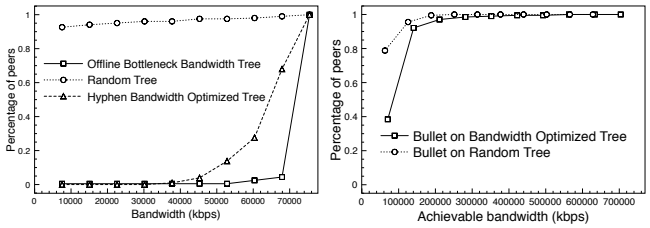
This section presents our evaluation of the benefits of using Hyphen. In what follows, we first describe our simulation setup and then evaluate the performance of various overlays built by Hyphen with different optimisation goals.

Evaluation setup. We evaluate the performance of Hyphen using the Sinalgo simulator [2]. Sinalgo acts in rounds, which we consider as our time unit. In each round, a node receives and sends messages from/to its direct neighbours. To approximate a real environment, we performed experiments on a simulated network based on approximations of a PlanetLab network. The adopted approximation covers a well-connected PlanetLab network with 205 nodes and 35918 links. This approximation is part of another contribution detailed in [15]. The resulting topology view includes several components details. In this evaluation, we retain the links' bandwidth and the latency. The bandwidth measured on this network links ranges in $[0, 100Mbps]$, while links latencies are in $[0, 24283ms]$. In both simulated networks, the peers' churn probabilities P_i and links' loss probabilities L_i are chosen uniformly at random from different predefined ranges and fixed for the duration of an experiment.

Bandwidth optimised tree for Bullet. A single tree is the basis of a large number of multicast protocols. As already explained, Hyphen could be used to build various type of trees. For the current evaluation, we chose to evaluate the construction of a tree optimising bandwidth to serve Bullet [21]. We name this tree the *Bandwidth Optimised Tree* (BOT). For this, we simulate the construction of our BOT and the construction of a random covering tree. On top of these trees we simulate the Bullet algorithm building a mesh dynamically during a streaming. In this experiment, we fix a maximum number of peering relationships (i.e., new links) part of the Bullet mesh to 2 for each peer and the *RanSub* epoch to 5 rounds.

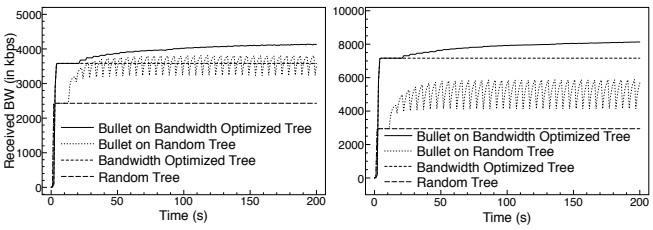
To evaluate the performance of BOT, we fix as a benchmark to our tree the one defined in [21] as the best possible bandwidth optimised tree and named *Bottleneck bandwidth tree*. In [21], the bottleneck bandwidth tree was used for performance comparison with Bullet and was defined offline based on global topological information. In that work, authors argued that it would be extremely difficult for any online tree-based algorithm to exceed the bandwidth achievable by the offline building algorithm. We agree with authors' argumentations. However, we believe that the BOT built by Hyphen is close to the Bottleneck bandwidth tree. Figure 3 (a) compares the bandwidth performance of the Bottleneck bandwidth tree and the BOT built by Hyphen. It shows the *Cumulative Distribution Function* (CDF) of the average achievable bandwidth at each node based on the BOT and based on our benchmark tree. It also shows the CDF of the average achievable bandwidth when using a random tree. Such a tree could be the one currently used by Bullet. By achievable bandwidth we mean the maximum rate a node can receive using these overlays. As we can see, the achievable bandwidth using our BOT, built in a scalable manner, is close to the achievable bandwidth when using the benchmark tree based on global knowledge. More interestingly, using our BOT peers may receive by far a better bandwidth than when using random tree as the one used by Bullet. This indicates a promising improvement of the Bullet streaming quality if relying on BOT. To measure this possible improvement, Figure 3 (b) shows the CDF of the maximum achievable bandwidth when performing Bullet on random tree and on our BOT. Since the Bullet mesh is built dynamically during a streaming, the achievable bandwidth measured here was captured a time t=100 during a streaming 3600kbps. From this figure, we observe that when defining Bullet on a random tree 80% of peers have an achievable bandwidth less than 70Mbps whereas, performing Bullet on our BOT only 40% are bounded to this same achievable bandwidth. In addition, we notice, in this same scenario, that the number of new peering relationships (i.e., Bullet mesh links) defined by Bullet on top of our BOT is much less (= 205 links) than the number defined on top of a random tree (= 380).

To show the improvement in terms of streaming quality, we measure the average of received bandwidth when performing a streaming on single covering trees and when performing the Bullet streaming solution. Figure 4 plots the received bandwidth when performing different streaming rates. As shown in Figure 4 the bandwidth performance of our BOT is higher than the performance of the random



(a) trees overlays (b) Bullet mesh at time=100 when streaming 3600kbps

Figure 3: CDF of maximum achievable bandwidth.



(a) Streaming 3600 kbps (b) Streaming 7200 kbps

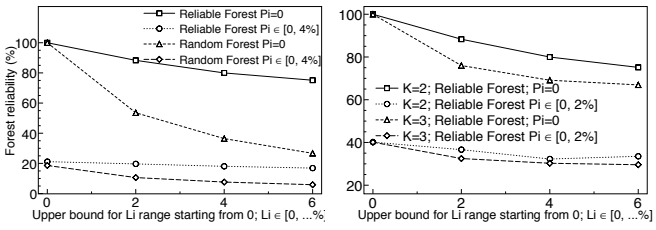
Figure 4: Achieved bandwidth over time for streaming over trees and Bullet mesh.

tree. While the received bandwidth using BOT scales as the required streaming rate increases, the received bandwidth when using a random tree does not reach the required rate. Regarding the Bullet behaviour, we can see that Bullet mesh has a more significant impact when using random tree as it permits to hide the missed messages on the random tree. On the other hand, BOT offers almost the required streaming and thus the cross links built by Bullet permit to completely hide any message misses.

Reliable forest for SplitStream. In this section, we investigate the reliability gain in forest overlay built using Hyphen with reliability optimisation. Our results show that Hyphen can build reliable overlays that enhance significantly the reliability of SplitStream when used instead of the forest overlay built without an optimisation focus. We measure the reliability of a forest as the average reliability over disjoint trees composing it. The reliability of each disjoint tree is the product of reliabilities of branches composing that tree computed iteratively using our *Upd* function defined in Section 5.1. Figure 5 shows the reliability of a forest overlay built with Hyphen optimising reliability and the reliability of a random forest built without optimisation. As shown in Figure 5 (a) in a reliable environment (L_i=0 and P_i=0), there is no difference between our reliable forest and the random forest since both of them provide 100% reliability. As soon as we inject unreliability to the environment configuration our reliable forest achieves better reliability than the random forest. This reliability advantage varies with the environment configuration. For the same peer churn probability P_i, this difference increases as the message loss probability L_i increases. Contrary, for the same message loss probability L_i, this difference decreases as the churn probability P_i increases.

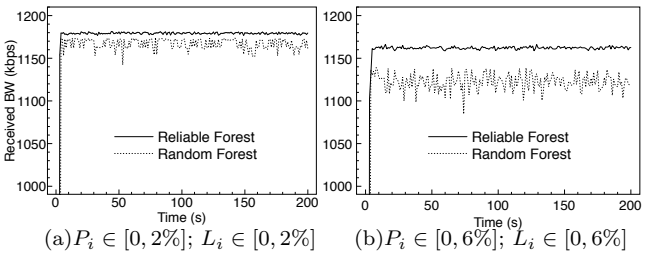
Regarding the number of stripes K in the forest, Figure 5 (b) shows the reliability of Hyphen reliable forest while varying K. From this figure, we notice that the forest reliability decreases when K increases. This reliability decrease is more important when peers are reliable (P_i=0). This is because the peers' unreliabilities have a larger impact on the overall reliability of the forest. This is simply because such overlay includes all peers and not all links of the network. Thus, when the peers are unreliable the forest reliability is improved slightly by selecting the most reliable links.

To show the advantage of reliable forest, we simulate a streaming through our reliable forest and through forest overlays built without optimisation focus, named *Random Forest*. In this streaming some interruptions can occur due to churn disconnecting temporarily the overlay or due to a link losing a message. The interruption risk in one path is proportional to the path reliability. To focus on reliability, in this evaluation we do not consider bandwidth limitation. In other words, no messages are missed due to bandwidth limitation. A message is lost only due to the unreliability of peers and links. Figure 6 shows the average received bandwidth when streaming 600kbps in each stripe of a forest overlay with 2 stripes while varying the peers unreliabilities P_i and links unreliabilities L_i. As it is noticeable, our reliable forest ensures a higher bandwidth as fewer interruptions, due to churn or to links losing messages, occur. The difference between received bandwidth gets larger when the components unreliabilities (P_i & L_i) increase (Figure 6 (b)), because our reliable forest is defined by selecting the most reliable paths by taking into account individual component unreliability.



(a) Forest with 2 stripes (K=2) (b) 2 stripes vs 3 stripes

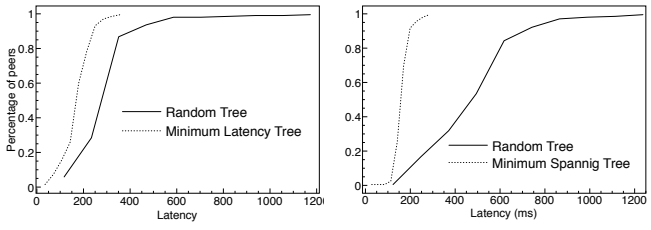
Figure 5: Forest reliability



(a)$P_i \in [0, 2\%]$; $L_i \in [0, 2\%]$ (b)$P_i \in [0, 6\%]$; $L_i \in [0, 6\%]$

Figure 6: Received bandwidth from a streaming of 600kbps on each stripe of a forest with two stripes (K=2).

Minimum latency multi-trees. In this section, we evaluate the latency advantage of a multi-tree overlay optimising latency. For this, we compare the delay experienced at each node when using a random multi-trees overlay and when using a minimum latency multi-trees built by Hyphen. Figure 7 plots the CDF of average delay that each peer experiences when included in one tree overlay (a) or when included in four distinct trees of a multi-trees overlay (b). In both single tree and multi trees overlays, the delay experienced by



(a) one tree topology (b) 4 trees topology

Figure 7: CDF of average delay in multi-trees

each peer when using Hyphen minimising latency is considerably lower than when being part of these overlays topologies built randomly. Note also that the advantage of using Hyphen optimising latency is more important when tested in other networks. Indeed, the simulated PlanetLab network represents a well connected graph with a large number of links. When investigating these links' latencies, we notice that almost 75% of nodes are connected to at least one link with a very low latency (20 ms). Thus, with a large number of low latency links, even a randomly constructed overlay can achieve a reasonable delay.

7. CONCLUSION

In this paper, we presented Hyphen, a middleware solution for constructing and maintaining various tree-like overlays. Hyphen aims at supporting the existing application-level multicast at two levels: simplification and quality improvement. The simplification is achieved by factoring out the construction and maintenance of *generic* overlay topologies. In addition, Hyphen aims at improving the quality of existing multicast solutions by building *generic* overlays following various optimisation focus. In this paper, three main optimisations are provided: minimise latency, maximise bandwidth and maximise reliability. Our experimental evaluation shows that Hyphen can construct overlays that significantly improve the quality of Bullet and SplitStream. In future work, we plan to extend Hyphen to construct other overlay topologies, e.g., mesh, and optimising other performance properties, e.g., peer degree. In addition, we want to explore a real-world deployment on the public Internet as part of a middleware system for P2P applications.

8. REFERENCES

[1] `http://java.net/projects/jxta`.

[2] `http://dcg.ethz.ch/projects/sinalgo`.

[3] Introduction to windows peer-to-peer networking. `http://technet.microsoft.com/en-us/library/bb457079.asp`.

[4] M. Allani, B. Garbinato, A. Malekpour, and F. Pedone. Quocast: A resource-aware algorithm for reliable peer-to-peer multicast. In *Proceedings of NCA*, 2009.

[5] M. Allani, B. Garbinato, and F. Pedone. Application layer multicast. In B. Garbinato, H. Miranda, and L. Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, chapter 9, pages 191–214. Springer, Mar. 2009.

[6] M. Allani, B. Garbinato, F. Pedone, and M. Stamenkovic. A gambling approach to scalable resource-aware streaming. In *Proceedings of SRDS*, pages 288–297, October 2007.

[7] M. Allani, J. Leitao, B. Garbinato, and L. Rodrigues. Rasm: Reliable algorithm for scalable multicast. In *Proceedings of PDP*, 2010.

[8] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proceedings of DSN*, 2007.

[9] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *Proceedings of SOSP*, 2003.

[10] Y. Chawathe, S. McCanne, and E. A. Brewer. RMX: Reliable multicast for heterogeneous networks. In *INFOCOM*, 2000.

[11] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of Sigmetrics*, 2000.

[12] M. Ferreira, J. Leitão, and L. Rodrigues. Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. In *Proceedings of SRDS*, 2010.

[13] P. Francis. Yoid: Extending the internet multicast architecture. Technical report, ACIRI, 2000.

[14] B. Garbinato, F. Pedone, and R. Schmidt. An adaptive algorithm for efficient message diffusion in unreliable environments. In *Proceedings of IEEE DSN*, 2004.

[15] T. Haddow, S. W. Ho, J. Ledlie, C. Lumezanu, M. Draief, and P. R. Pietzuch. On the feasibility of bandwidth detouring. In *PAM*, pages 81–91, 2011.

[16] N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications*, pages 879–894, 2003.

[17] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking*, 2003.

[18] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. O'Toole. Overcast: reliable multicasting with on overlay network. In *OSDI'00*.

[19] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of ESOA*, 2005.

[20] K. Kim, S. Mehrotra, and N. Venkatasubramanian. Farecast: Fast, reliable application layer multicast for flash dissemination. In *Middleware*, 2010.

[21] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of SOSP*, 2003.

[22] M. Kwon and S. Fahmy. Path-aware overlay multicast. *Computer Networks*, pages 23–45, 2005.

[23] J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Proceedings of SRDS*, 2007.

[24] B. Li, J. Guo, and M. Wang. ioverlay: a lightweight middleware infrastructure for overlay application implementations. In *Proceedings of Middleware*, 2004.

[25] J. Liebeherr and T. K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Proceedings of NGC*, 1999.

[26] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of SOSP*, 2005.

[27] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 2005.

[28] A. Malekpour, F. Pedone, M. Allani, and B. Garbinato. Streamline: An architecture for overlay multicast. In *Proceedings of NCA*, 2009.

[29] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. Mosaic: unified declarative platform for dynamic overlay composition. In *Proceedings of CoNEXT*, 2008.

[30] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of NSDI*, 2004.

[31] G. Tan, S. A. Jarvis, and D. P. Spooner. Improving the fault resilience of overlay multicast for media streaming. In *DSN*, 2006.

[32] C. Tang and C. Ward. GoCast: Gossip-enhanced overlay multicast for fast and dependable group communication. In *Proceedings of DSN*, 2005.

[33] Y. Tian, H. Shen, and K.-W. Ng. Improving reliability for application-layer multicast overlays. *IEEE Trans. Parallel Distrib. Syst.*, 2010.

[34] P. F. V. Venkataraman and J. Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer. In *Workshop on Peer-to-Peer Systems*, 2006.