Resource-Aware Multimedia Content Delivery: A Gambling Approach

Mouna Allani[1], Benoît Garbinato[1], Fernando Pedone[2]

[1]UNIL, Switzerland [2]USI, Switzerland

Abstract

In this paper, we propose a resource-aware solution to achieving reliable and scalable stream diffusion in a probabilistic model, i.e., where communication links and processes are subject to message losses and crashes, respectively. Our solution is resource-aware in the sense that it limits the memory consumption, by strictly scoping the knowledge each process has about the system, and the bandwidth available to each process, by assigning a fixed quota of messages to each process. We describe our approach as *gambling* in the sense that it consists in accepting to give up on a few processes sometimes, in the hope to better serve all processes most of the time. That is, our solution deliberately takes the risk not to reach some processes in some executions, in order to reach every process in most executions. The underlying stream diffusion algorithm is based on a tree-construction technique that dynamically distributes the load of forwarding stream packets among processes, based on their respective available bandwidths. Simulations show that this approach pays off when compared to traditional gossiping, when the latter faces identical bandwidth constraints.

1 Introduction

Reliable stream diffusion under constrained environment conditions is a fundamental problem in large-scale multimedia content delivery. In this context, the efficiency of a given content delivery solution directly depends on the performance of its underlying multicast protocol. Environment conditions are typically constrained by the reliability and the capacity, usually limited, of its components. Nodes and communication links can fail, unexpectedly ceasing their operation and dropping messages, respectively. Moreover, real-world deployment does not offer nodes and links infinite memory and infinite bandwidth. Therefore, realistic solutions should use local storage and inter-node communication sparingly, and account for node crashes and message losses.

In this paper, we investigate the problem of reliable stream diffusion in unreliable and constrained environments from a novel angle. Our approach is probabilistic: with high probability, all consumers will be reached and deliver all information addressed to them; however, there is no guarantee that this will happen. Differently from previous probabilistic algorithms found in the literature, we resort to a "gambling approach," which deliberately penalizes a few consumers in rare cases, in order to benefit most consumers in common cases. We show experimentally that the approach pays off in that it outperforms traditional gossip-based algorithms when subject to similar environment constraints.

The key idea of our solution is to stream multimedia content according to a global propagation graph. This graph approximates a global tree aiming at the maximum reachability and efficient use of the available bandwidth. The approach is completely decentralized: nodes build propagation trees, which we call *Maximum Probability Trees* (MPTs), autonomously. Several MPTs are dynamically composed to achieve a global graph reaching most (hopefully all) consumer nodes. This solution is scalable and based on a composition of *local optimums*, i.e., each MPT ensures the maximum probability of reaching all processes in its subgraph when subject to bandwidth constraints. MPTs are composed in a manner that respects bandwidth constraints, and the MPT construction is fully parameterized. Nodes are free to define the scope of their local knowledge, from direct neighborhood to the entire network. The scope of each process can be defined according to its local constraints (e.g., processing power, memory capacity).

Besides discussing a new reliable stream diffusion algorithm, we also show that it can be implemented in a very modular way, lending itself to real deployment. Our solution consists in decomposing the problem of reliable stream diffusion into sub-problems. This separation of concerns gives rise to an architecture composed of five layers.

The remainder of this paper is organized as follows. In Section 2 we introduce the system model and define the problem that motivates this work. Section 3 describes our reliable streaming solution based on a tree-construction technique. Section 4 describes a performance evaluation of our approach, including an analysis of the costs and benefits of gambling. We discuss related work in Section 5. Finally, in Section 6 we summarize our findings and conclude with some final remarks.

2 Scalable Resource-Aware Streaming

Stream diffusion is a typical 3-step scenario: (1) the producer breaks the outgoing stream into elemental messages (stream packets) and multicasts them to interested consumers, (2) intermediate nodes route these messages to the consumers, and (3) each consumer recomposes the received messages into a coherent incoming stream. This is depicted in Figure 1. In a resource-

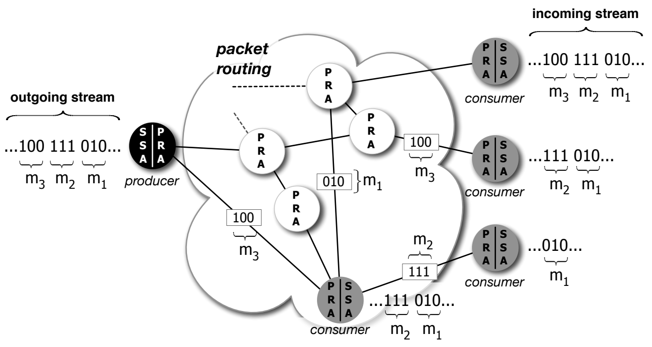


Figure 1: Multimedia Content Delivery – Stream diffusion scenario

constrained environment, the main challenge then consists in routing stream messages in a way that efficiently uses available resources.

2.1 Basic system model

We consider an asynchronous distributed system composed of processes (nodes) that communicate by message passing. Our model is probabilistic in the sense that processes can crash and links can lose messages with a certain probability. More formally, we model the system's topology as a connected graph $G = (\Pi, \Lambda)$, where $\Pi = \{p_1, p_2, ..., p_n\}$ is a set of processes of size n, and $\Lambda = \{l_1, l_2, ...\} \subseteq \Pi \times \Pi$ is a set of bidirectional communication links. Process crash probabilities and message loss probabilities are modeled as *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 one computation step and L_j is the probability that link l_j loses a message during one communication step.

2.2 Problem statement

Intuitively, the main question addressed in this paper is the following: *how can we make stream messages reach all consumers with a high probability, in spite of unreliable processes and links, and of the limited resources (e.g., bandwidth) available to each process?*

Formally, the *limited resources constraint* is modeled as $Q = (q_1, q_2, ..., q_n)$, the set of quotas associated to processes in the system. Each *individual quota of messages* q_i represents the number of messages process p_i able to send in order to forward a single stream packet. A quota may represent a set of physical constraints related to the limited hardware resources or a dedicated percentage of these resources fixed by the peer itself. This percentage captures the fact that the user behind a peer can voluntary limit the resources dedicated to the P2P streaming service. In other words, a quota is a translation of both the percentage of hardware resources a peer is willing to dedicate to forward a stream packet and the upload limit of the ISP of the peer, which might be further limited by the percentage of that bandwidth the peer is willing to dedicate to the streaming service. By extending the basic system model presented earlier, we then can say that the tuple $S = (\Pi, \Lambda, C, Q)$ completely defines the system considered in this paper.

In order to take into account *processing and memory constraints*, we further assume that each process has only a *partial* view of the system, meaning that its routing decisions can only be based on incomplete knowledge. Formally, the limited knowledge of process p_i is modeled with *distance* d_i, which defines the maximum number of links in the shortest path separating p_i from any other node in its known subgraph. Distance d_i implicitly defines the partial knowledge of p_i as scope $s_i = (\Pi_i, \Lambda_i, C_i, Q_i)$, with $\Pi_i \subseteq \Pi$, $\Lambda_i \subseteq \Lambda$, $C_i \subseteq C$, and $Q_i \subseteq Q$. In the remainder of this paper, any graph comprised of processes and links should be understood as also including the corresponding configuration and quota information.

Based on the above definitions, we can now restate the problem we address in this paper more succinctly: *given its limited scope s_i, how should process p_i use its quota q_i in order to contribute to reach all consumers with a high probability?*

3 A Gambling Approach

In the absence of any constraints on resources, making stream messages reach all processes with a high probability is quite easy, typically via some generous gossiping (or even flooding) algorithm. In a large-scale resource-constrained system, however, such a solution is not realistic.

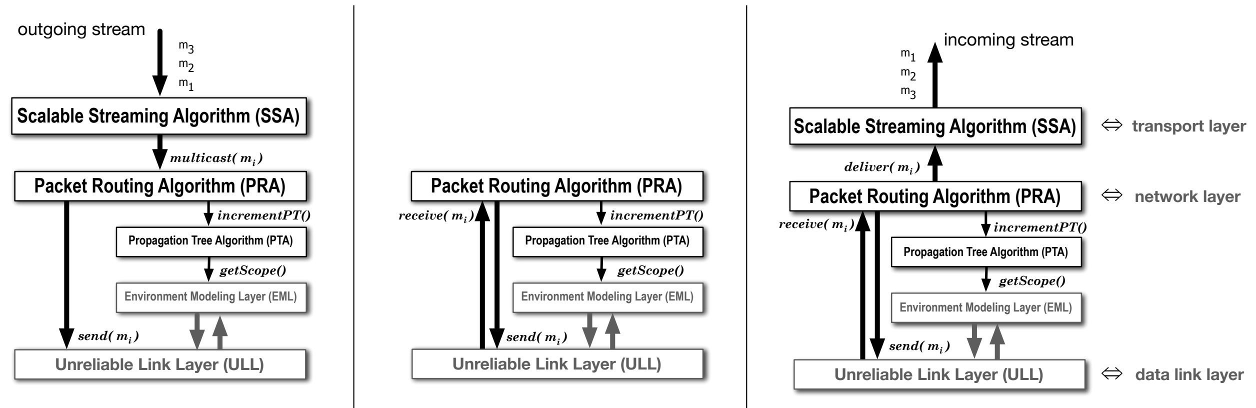
3.1 Diffusion trees as starting point

The starting point of our approach can be found in [1], where we proposed an algorithm to efficiently diffuse messages in a probabilistically unreliable environment. Intuitively, the solution consists in building a spanning tree that contains the most reliable paths connecting all processes, using a modified version of Prim's algorithm [32]. The algorithm is also somehow resource-aware in that it tries to minimize the number of messages necessary to reach all processes with a given probability.

This algorithm, however, does not limit the bandwidth: when asking the algorithm to diffuse a message with a high probability in a very unreliable environment, the number of messages tends to explode. Furthermore, this solution does not limit memory consumption either: in order to achieve optimality, it requires a complete knowledge of the system topology and of the failure probabilities associated to links and processes. Informally, the approach presented hereafter consists in building a diffusion graph that exhibits properties similar to that of [1], while taking into account strict constraints on resources (bandwidth, memory, etc). As presented in Section 2, these contraints are modeled via q_i and s_i, respectively the limited quota and the limited scope available at each process p_i.

As soon as we face resource constraints, we have to make difficult decisions. In the context of this paper, this observation translates into deciding how high the risk we are willing to take is, in order to increase our chances to reach all consumers. More specifically, the question we ask ourselves is the following: does it pay off to take the risk to sacrifice a few consumer processes in some executions, in order to reach every process in most executions? As we shall see in Section 4, when comparing the performance of our solution to that of a typical gossiping approach, the answer is clearly *yes*.

Intuitively, our approach consists in having processes make bold decisions, in spite of their limited view of the system (scope), in the hope to better use the available resources (quota). That is, along the paths from the producer to the consumers, a process p_i may decide to build a local propagation tree based on its limited scope s_i in order to maximize the probability to



(a) producer & router (b) pure router (c) consumer & router

Figure 2: A layered architecture

reach everybody in s_i.[1] In building its local propagation tree, p_i also decides how processes in s_i should use their quotas. Since these decisions can be made concurrently, process p_i has no guarantee that processes in s_i will actually follow its decisions. As we shall see in Section 4, this approach can lead to some (fairly rare) executions in which some processes are never reached. Experiments show however that the benefits of taking such a risk pays off in most executions.

3.2 Solution overview

Our solution is based on the five-layer architecture pictured in Figure 2. The top layer represents a standard stream fragmentation layer. It executes the *Scalable Streaming Algorithm (SSA)*, is responsible for breaking the outgoing stream into a sequence of messages on the producer side, and for assembling these messages back into an incoming stream on the consumer side. Roughly speaking, this layer corresponds to the *Transport* layer in the OSI model [2]. The SSA layer then relies on the *Packet Routing Algorithm (PRA)*, which is responsible for routing stream messages through a *propagation graph* covering the whole system; this layer corresponds to the *Network* layer in the OSI model. This propagation graph results from the spontaneous aggregation of various *propagation trees* concurrently computed by some intermediate routing processes defined as responsible for this task. As suggested by Figure 2, producers and consumers execute both the SSA and PRA layers, while pure routing processes only execute the PRA layer. The responsibility for building propagation trees is delegated to the *Propagation Tree Algorithm (PTA)*, which in turn relies on the partial view delivered by the *Environment Modeling Layer (EML)*. The latter relies on Bayesian inference to approximate the environment within distance d_i of each process p_i. Explaining how the environment modeling actually works falls beyond the scope of this paper and can be found in [1]. Finally, the *Unreliable Link Layer (ULL)* allows each process p_i to send messages to its direct neighbors in a probabilistically unreliable manner. This layer corresponds to the *Data Link* layer of the OSI model.

[1]The actual criteria that determines whether p_i will make such a decision or not is explained later.

3.3 Scalable Streaming Algorithm (SSA)

The scalable streaming solution, presented in Algorithm 1, is fairly straightforward. On the producer side, as long as some data is available from the outgoing stream (line 6), the algorithm reads that data, builds up a message containing it and multicasts the message using the *multicast*() primitive of the PRA layer (lines 7 to 10). On the consumer side, upon receiving a message from PRA (line 11), the algorithm writes the data contained in that message to the incoming stream, provided that the message is not out of sequence (lines 12 to 14). Because of the probabilistic nature of our environment, messages can indeed be received out of sequence, in which case they are simply dropped. This is the standard way to handle out-of-sequence packets when streaming realtime data, such as audio or video streams. Note that this strategy can be easily improved by a simple local buffering mechanism in order to deal with jitter and out-of-order messages.

Algorithm 1 Scalable Streaming Algorithm at p_i

```
 1: uses: PRA
 2: initialization:
 3:   $nextSeq \leftarrow 1$
 4:   $lastSeq \leftarrow 0$

 5: To multicast some $outgoingStream$ to a set of $consumers$:
 6:   while not $outgoingStream.eof()$ do
 7:     $m.data \leftarrow outgoingStream.read()$
 8:     $m.seq \leftarrow nextSeq$
 9:     $nextSeq \leftarrow nextSeq + 1$
10:     PRA.multicast($m, consumers$)

11: upon PRA.deliver($m$) do
12:   if $m.seq > lastSeq$ then
13:     $incomingStream.write(m.data)$
14:     $lastSeq \leftarrow m.seq$
```

3.4 Packet Routing Algorithm (PRA)

The packet routing solution, presented in Algorithm 2, consists in disseminating stream messages through a *propagation graph* generated in a fully decentralized manner. This propagation graph actually results from the spontaneous aggregation of several *propagation trees*. Each propagation tree is in turn the result of an incremental building process carried out along the paths from the producer to the consumers. It is important to note however that the aggregated propagation graph itself might well not be a tree.

On the producer. The routing process starts with producer p_i calling the *multicast*() primitive (line 4). As a first step, p_i asks the PTA layer to build a first propagation tree *pt*, using the *incrementPT*() primitive (line 5). This primitive is responsible for incrementing the propagation tree passed as argument, using the scope of the process executing it (here p_i). Since p_i is the producer, the initial propagation tree passed as argument is simply composed of p_i and its associated information (failure probability P_i and quota q_i). As discussed in Section 3.5, the returned propagation tree *pt* maximizes the probability to reach everybody in scope s_i, based on available quotas. Process p_i then calls the *optimize*() primitive, passing it *pt* (line 6). This primitive is discussed in details in Section 3.7. At this point, all we need to know is that it

Algorithm 2 Packet Routing Algorithm at p_i

```
1: uses: PTA, ULL, EML
2: initialization:
3:     $r \leftarrow ...$

4: procedure $multicast(m)$
5:     $pt \leftarrow$ PTA.incrementPT( $(\{p_i\}, \emptyset, \{P_i\}, \{q_i\})$ )
6:     $\vec{m} \leftarrow optimize(pt)$
7:     $propagate(m, pt, p_i, \vec{m})$

8: upon ULL.receive($m, p_k, pt, \vec{m}$) do
9:     if EML.distance($p_k, p_i$) $\geq r$ then
10:        $pt \leftarrow$ PTA. incrementPT($pt$)
11:        $\vec{m} \leftarrow optimize(pt)$
12:        $propagate(m, pt, p_i, \vec{m})$
13:    else
14:        $propagate(m, pt, p_k, \vec{m})$
15:    if $p_i$ is interested in $m$ then
16:        SSA.deliver($m$)

17: procedure $propagate(m, pt, p_k, \vec{m})$
18:    for all $p_j$ such that link $(p_i, p_j) \in E(pt)$ do
19:        repeat $\vec{m}[j]$ times :
20:            ULL.send($m, p_k, pt, \vec{m}$) to $p_j$
```

returns a propagation vector $\vec{m}$ indicating, for each link in pt, the number of messages that should be sent through that link in order to maximize the probability to reach everybody in scope s_i. Finally, p_i calls the *propagate*() primitive (line 7), which simply follows the forwarding instructions computed by *optimize*(). That is, it sends stream message m, together with some additional information, to p_i's children in pt. As we shall see below, this additional information is used throughout the routing process to build up the propagation graph.

On the consumer. When a consumer p_i receives message m, together with the aforementioned information (line 8), it has first to decide whether to increment pt before further propagating m (lines 10 to 12), or to simply follow the propagation tree pt it just received (line 14). The propagation tree pt should be incremented if and only if the distance that separates p_i from p_k, the process that last incremented pt, is equal to $r \leq d_k$, the *increment rate*. In such a case, p_i is said to be an *incrementing node*.

Intuitively, r defines how often a propagation tree should be incremented as it travels through the propagation graph. The latter then spontaneously results from the concurrent and uncoordinated increments of propagation trees finding their ways to the consumers. Finally, process p_i delivers message m to the SSA layer only if it is interested in it (lines 15 and 16). If this is not the case, process p_i is merely a router node.

3.5 Propagation Tree Algorithm (PTA)

The solution to increment propagation trees is encapsulated in the *incrementPT*() primitive, presented in Algorithm 3. This primitive takes a propagation tree pt as argument and increments it if needed, i.e., if something changed in the environment of p_i or if pt is different from the propagation tree that was last incremented (line 8). The conditional nature of this increment is motivated by performance and resources concerns: during stable periods of the system, propagation trees remain unchanged, cutting down the processing load of incrementing nodes. To get an up-to-date view of its surrounding environment, p_i calls the *getScope*() primitive provided by EML (line 7).

To build local tree lpt_i, process p_i first builds a *Maximum Probability Tree (MPT)*, using the *mpt*() primitive (line 11). Details about the notion of maximum probability tree, and primitive *mpt*(), are provided in Section 3.7. Briefly, MPT maximizes the probability to reach every process within a given scope, by taking into account not only the intrinsic reliability of processes and links in scope s_i, but also the individual quotas available to processes in s_i. Note that primitive *mpt*() increments *pt* as a whole (see discussion below), whereas Algorithm 3 is in fact only interested in the subtree rooted at p_i (line 12). This subtree is precisely the local tree lpt_i.

Algorithm 3 Propagation Tree Algorithm at p_i

```
1:  uses: EML
2:  initialization:
3:     $lpt_i \leftarrow \emptyset$
4:     $pt_i \leftarrow \emptyset$
5:     $s_i \leftarrow \emptyset$

6:  function $incrementPT(pt)$
7:     $s \leftarrow$ EML.getScope( )
8:     if $pt_i \neq pt \;\vee\; s_i \neq s$ then
9:        $pt_i \leftarrow pt$
10:       $s_i \leftarrow s$
11:       $myMpt \leftarrow mpt(s_i, pt_i)$
12:       $lpt_i \leftarrow$ subtree of $myMpt$ with $p_i$ as root
13:    return $pt \cup lpt_i$
```

3.6 The gambling effect.

Intuitively, the approach taken by the *mpt*() primitive consists in augmenting *pt* with the best branches in scope s_i, even if some of these branches are not downstream from p_i. These latter branches are said to be *concurrent branches.* This approach somehow consists in taking the risk to exclude some consumers from the propagation graph by accident. Process p_i has indeed no way to inform processes located along concurrent branches about its incremental decisions, and has no guarantee that incremental decisions will be taken coherently with respect to each other. In order to partially mitigate this risk, Algorithm 3 merges the local tree with the original propagation tree passed as argument (line 13), rather than directly returning the maximum reliability tree.

Execution example. Figure 3 illustrates the incrementing of the propagation tree on a simple example. In this scenario, the distance defining the scope and the increment rate r are the same for all processes and equal to 2. Process p_1, the producer, builds a first *propagation tree* pt_1 covering its scope s_1; this tree is pictured in Figure 3 (a) using bold links. All nodes in pt_1 that are at a distance $r = 2$ from p_1 are *incrementing nodes*, which means they have to increment pt_1 when they receive it. Process p_3 being such a node, it calls the *mpt*() function, passing it pt_1 and its scope s_3. This function adds the dashed links pictured in Figure 3 (a)

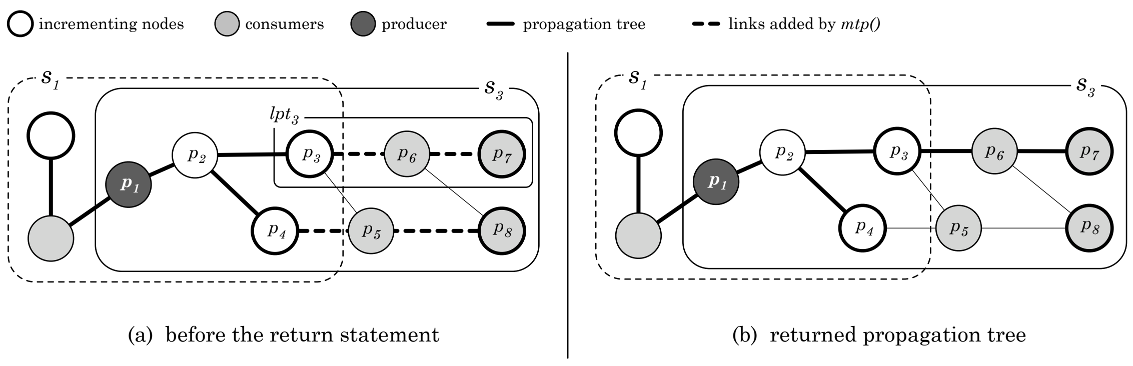


Figure 3: Propagation tree increment

to pt_1 and returns the resulting Maximum Probability Tree (MPT); this MPT contains the local propagation tree rooted at p_3, i.e., lpt_3. The latter is then extracted from the MPT, merged with the initial propagation tree pt_1 and returned. Figure 3 (b) pictures the new propagation tree resulting from the above increment process.

3.7 Maximum Probability Tree (MPT)

The concept of *Maximum Probability Tree (MPT)* is at the heart of our approach, as it materializes the risk taken during the construction of the propagation graph. Intuitively, an MPT maximizes the probability to reach all processes within a given scope by optimally using the quotas of these processes. Before describing how the $mpt()$ function given in Algorithm 4 builds up an MPT, we first recall the notions of *reachability probability* and *reachability function*.

Reachability probability. The *reachability function*, denoted $R()$, computes the probability to reach all processes in some propagation tree T with configuration $C(T)$, given a vector $\vec{m}$ defining the number of messages that should transit through each link of T. We then define the probability returned by $R()$ as T's *reachability probability*. Equation 1 below proposes a simplified version of the reachability function borrowed from [1] — this version assumes that only links can fail by losing messages with a given probability, whereas processes are assumed to be reliable.[2]

$$R(T,\vec{m}) \;=\; \prod_{j=1}^{|\vec{m}|} 1 - L_j^{m[j]} \; where \; L_j \in C(T) \qquad (1)$$

Using $R()$, we then define the $maxR()$ function presented in Algorithm 4 (lines 8 to 10), which returns the maximum reachability probability for T. To achieve this, $maxR()$ first calls the $optimize()$ function in order to obtain a vector $\vec{m}$ that optimally uses the quotas available to processes in T. It then passes this vector, together with T, to $R()$ and returns the corresponding reachability probability.

The $optimize()$ function iterates through each process p_s in T and divides individual quota q_s in a way that maximizes the probability to reach direct children of p_s (line 14 to 20). For

[2]Note that this simplification causes no loss of generality; see [1] for details.

Algorithm 4 MPT | Building Process

```
function $mpt(S, T)$
    while $V(S) \not\subseteq V(T)$ do
        $O \leftarrow \{l_{j,k} \mid l_{j,k} \in E(S) \ \wedge \ p_j \in V(T) \ \wedge \ p_k \in V(S) - V(T)\}$
        let $l_{u,v} \in O$ such that $\forall l_{r,s} \in O$ :
        $maxR(T \ \cup \ l_{u,v}) \ \geq \ maxR(T \ \cup \ l_{r,s})$
            $T \leftarrow T \ \cup \ l_{u,v}$
    return $T$

function $maxR(T)$
    $\vec{m} \leftarrow optimize(T)$
    return $R(T, \vec{m})$

function $optimize(T)$
    let $\vec{m} : \forall l_j \in E(T)$, $\vec{m}[j]$ is the number of messages to be sent through link $l_j$
    $\vec{m} \leftarrow (1, 1, \cdots, 1)$
    for all $p_s \in V(T)$ do
        let $\Lambda_s \subset E(T)$ : $l_k \in \Lambda_s \Rightarrow (p_s, p_k) \in E(T)$
        if $\mid \Lambda_s \mid \ > \ q_s$ then
            return $(0, 0, \cdots, 0)$
        while $\sum_{l_k \in \Lambda_s} \ \vec{m}[k] < q_s$ do
            let $\vec{m}_u$ : $(l_u \in \Lambda_s) \wedge (\forall_{t \neq u} \ \vec{m}_u[t] = \vec{m}[t]) \wedge (\vec{m}_u[u] = \vec{m}[u] + 1) \wedge (R(T, \vec{m}_u) - R(T, \vec{m})$ is max)
            $\vec{m} \leftarrow \vec{m}_u$
    return $\vec{m}$
```

this, function *optimize*() allots messages of q_s one by one, until all messages have been allocated (line 18 to 20). That is, in each iteration step it chooses the outgoing link l_u from p_s that maximizes the gain in probability to reach all p_s's children in T, when sending one more message through l_u (line 19). When all individual quotas have been allocated, *optimize*() returns a vector $\vec{m}$ that provides the maximum reachability probability when associated with T.

MPT building process. We now have all the elements needed to present the MPT building process carried out by *mpt*(), given a scope S and an initial propagation tree T. This function simply iterates until all processes in S but not in T have been linked to T, i.e., it only stops when T covers the whole scope S (line 2 to 6). In each iteration step, the *mpt*() function then adds the link that produces a new tree exhibiting the maximum reachability probability (line 5).

Execution example. Figures 4 to 6 illustrate the MPT building process on a simple example. In this example, the initial tree T is composed of only process p_1 and S is the scope of p_1, i.e., $S = s_1$. During the first iteration step, the algorithm simply chooses the most reliable link, i.e., link $l_{1,2}$ with failure probability $L_{1,2} = 0.2$. At this point, it means that the entirety of p_1's quota has been allocated to reach p_2. In this example, the quota is identical for all processes and equal to 3, i.e., $\forall p_i : q_i = 3$.

At the beginning of the second step, the algorithm faces two alternatives: either adding link $l_{1,3}$ and splitting the quota of p_1 between links $l_{1,2}$ and $l_{1,3}$, or adding link $l_{2,4}$ and using the entirety of q_2, the quota of p_2, to reach p_4. These two alternatives are pictured in Figure 5 as trees T' and T'' respectively.

Based on the result of function $maxR()$, the algorithm chooses to keep T'', since it is the tree that offers the maximum probability to reach everybody. Note however that this decision implies

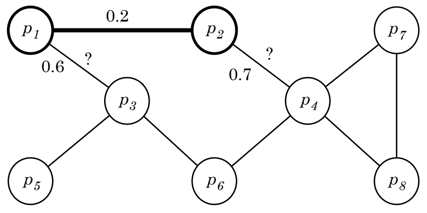


Figure 4: Resulting tree after the first iteration step

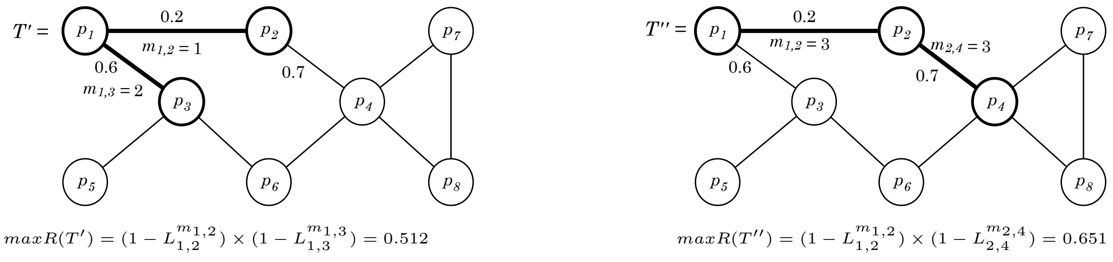


Figure 5: Alternative trees during the second iteration step

adding link $l_{2,4}$ rather than link $l_{1,3}$, although the latter is more reliable. Figure 6 pictures the final Maximum Probability Tree returned by function $mpt()$.

4 Performance evaluation

The performance of our scalable algorithm was evaluated through a simulation model. For simplicity, we only considered link failures, while assuming that processes are reliable, i.e, $\forall p_i : P_i = 0$. As mentioned in Section 3.7, this does not compromise the generality of our approach. We performed experiments with processes organized in various topologies: we started from a ring where each process had two neighbors and then incrementally augmented the number of neighbors until reaching a connectivity of 20 neighbors per process. This provided a spectrum of possibilities for the evaluations, starting with a worst-case topology with respect to process distances (i.e., the ring), and gradually reducing the mean distance between processes in the system by adding more links. Unless mentioned otherwise, we assumed topologies with 100 processes.

To facilitate the evaluation, we set the scope to be the same for all processes during the execution, i.e., $\forall p_i : d_i = d$. To avoid regular network configurations, we then defined 20% of

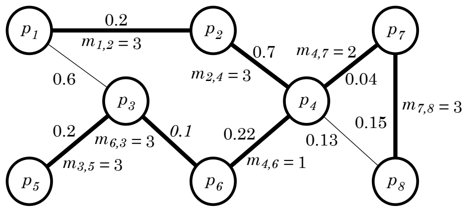


Figure 6: Final Maximum Probability Tree

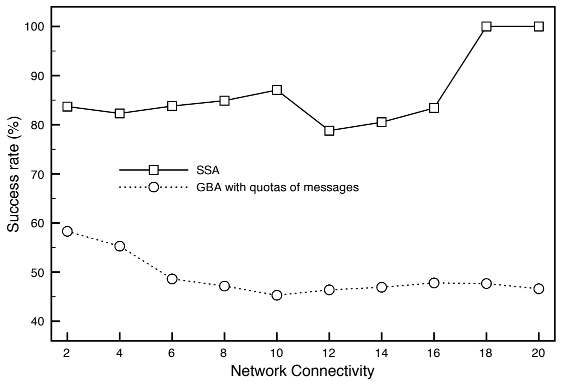


Figure 7: SSA *vs.* GBA with quotas – Success rate

processes to be hubs. A *hub* has twice the quota of a normal process and is connected to its neighbors through highly reliable links, i.e., we set the message loss probability of these links to 10^{-4}. Our performance evaluation consists in measuring the success rate of 1000 distinct executions. We consider an execution to be a success when the multicasted stream packet reaches all nodes in the system, i.e., the success rate is precisely what the notion of reachability probability tries to capture.

4.1 Benefits of gambling

Multicast protocols fall into two categories, those based on structured information dissemination such as our *Scalable Streaming Algorithm* (SSA), and those based on unstructured information dissemination, typically the case of gossip-based protocols. To measure the benefit of our gambling approach, we compare SSA with a typical *Gossip-Based Algorithm* (GBA), modified to implement the notion of individual quota: to propagate an incoming message m, the algorithm repeats the following two steps *until exhausting its quota*: (1) randomly choose a neighbor among those that did not yet acknowledge m and (2) send m to those neighbors. For the comparison, we then set the quota to 5 and the failures probability of each link[3] to a random value within $[0.05, 0.55]$. As for specific parameters of SSA, we set the scope to 5 and the increment rate to 2.

Figure 7 shows the evolution of the success rate of SSA and GBA respectively, when varying the network connectivity. As we can see, the success rate of GBA decreases as the connectivity increases. This is due to the fact that each process randomly uses its quota of messages, without taking into account the reliability of links. Indeed, as the connectivity increases, it becomes more and more important to maximize the impact of each message on the overall reachability probability.

For SSA on the contrary, the success rate tends to increase with the network connectivity because SSA has a larger choice of links when computing local Maximum Probability Trees (MPTs), and thus more chances to build a global propagation graph with a favorable reachability probability. Furthermore, even if some processes have a number of neighbors that exceeds their quota, our approach still tries to maximize the overall reachability probability by adapting the number

[3]To be more precise: each link that is *not attached to a hub.*

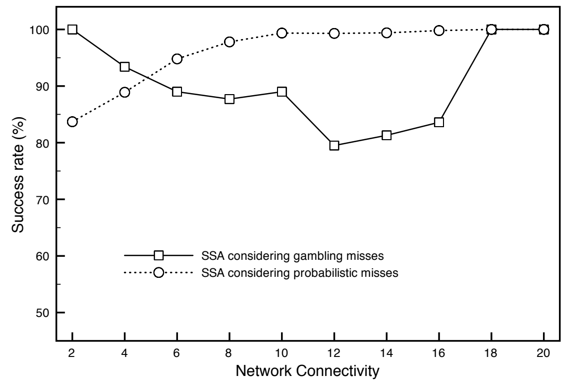


Figure 8: Influence of probabilistic misses and gambling misses on SSA's success rate

of children of each process to its quota. As shown in Figure 7, this has a significant impact on the actual success rate. For a connectivity of 20 for example, which is 4 times higher than the quota used in our experiments, the success rate is close to 100. In this figure however, we can also see a drop of the success rate for connectivities between 10 and 16. As discussed hereafter, this drop constitues the costs of gambling.

4.2 Cost of gambling

To evaluate the cost of our gambling approach, we introduce the notion of a *missed execution* of our algorithm. Such an execution, also simply called a *miss*, is one where at least one node in the system never received the multicast packet. We can further categorize such misses as either *probabilistic misses* or *gambling misses*. Probabilistic misses are caused by unreliable links sometimes losing messages, i.e., they are due to the probabilistic nature of the model we consider. Gambling misses on the other hand happen when the *effective* propagation graph does not cover the whole system. An effective propagation graph results from the aggregation of effectively followed *propagation trees*.

In Figure 8, we show how probabilistic misses and gambling misses influence the success rate of our algorithm, i.e., the two curves presented in this figure result from the decomposition of the SSA curve presented in Figure 7. Considering probabilistic misses, we can observe that as the connectivity increases, the probability of reaching all nodes also increases. This is not surprising, since as the connectivity increases, the number of links increases and the algorithm has a larger choice of links when computing MPT and thus more chances to get an MPT with a favorable reachability probability. For gambling misses on the contrary, as the connectivity increases, misses due to the structure of the effective propagation graph become more frequent because the algorithm has a larger choice of links, which induces a higher risk to make contradictory decisions when building distinct propagation trees. However, when reaching a high connectivity (12 links or more in our example), gambling misses become less frequent because the scope of each process becomes close to the whole system.[4]

[4]When the scope covers the whole system, the propagation graph corresponds to the MPT built by the producer and covering the whole system.In this case there is no gambling involved.

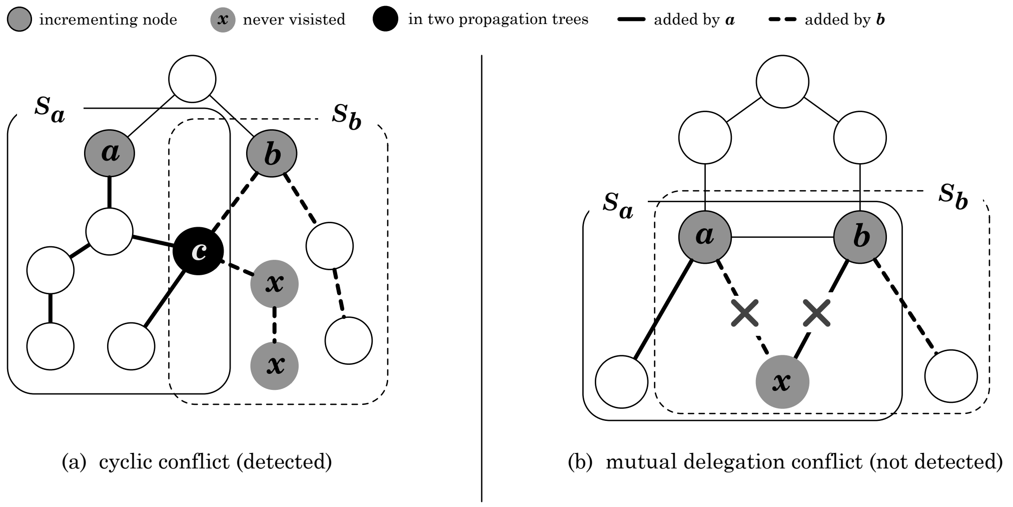


Figure 9: Conflicts causing gambling misses

Gambling cost mitigation The good news is that many cases of gambling misses are detectable and can be mitigated via a simple countermeasure, which leads to a few nodes exceeding their quotas. As discussed in Section 2.2, we assume that a quota of a node is not defined as the whole node propagation capacity. It can represent either a part of its capacity or the percentage of resources the peer allocated to the streaming service. As we just saw, a gambling miss occurs when the resulting effective propagation graph does not cover the whole system. Such misses can be caused by two types of conflicting situations, pictured in Figure 9.

A *cyclic conflict*, illustrated in Figure 9(a), is caused by the inclusion of some node c into two distinct propagation trees. When c receives a propagation tree, it uses its quota to propagate the packet in that tree, not knowing that a second tree will reach it. So, when the second propagation tree reaches c, the absence of remaining quota can cause some descendants of c in the second tree to never be reached. In Figure 9(a), node c receives two conflicting propagation trees, first one computed by node a and then one computed by node b. As a consequence, nodes below c in the tree computed by b might never be reached. It is easy to see that upon reception of the second tree, c is able to detect the conflict and to apply the countermeasure described hereafter.

A *mutual delegation conflict*, illustrated in Figure 9(b), is caused by contradictory decisions about how to include a given node, when incrementing two distinct propagation trees. In Figure 9 (b), node a decides to delegates the task of reaching node x to node b, while b decides to delegate the task of reaching x to a. As a consequence, node x will never be reached. Because incrementing nodes do not inform each other about their respective incrementing decisions, the mutual delegation conflict is not detected.

Cyclic conflict countermeasure. As already suggested, we can mitigate cyclic conflicts by occasionally having some nodes exceed their quotas. It is interesting to note however that the detection of a cyclic conflict by some node c does necessarily imply that some nodes might not be reached. More precisely, there exists two independent cases that require node c to exceed its quota. The first case is straightforward and occurs when a descendant node of c in the second propagation tree is not in the first propagation tree. This case is formalized by Condition 2

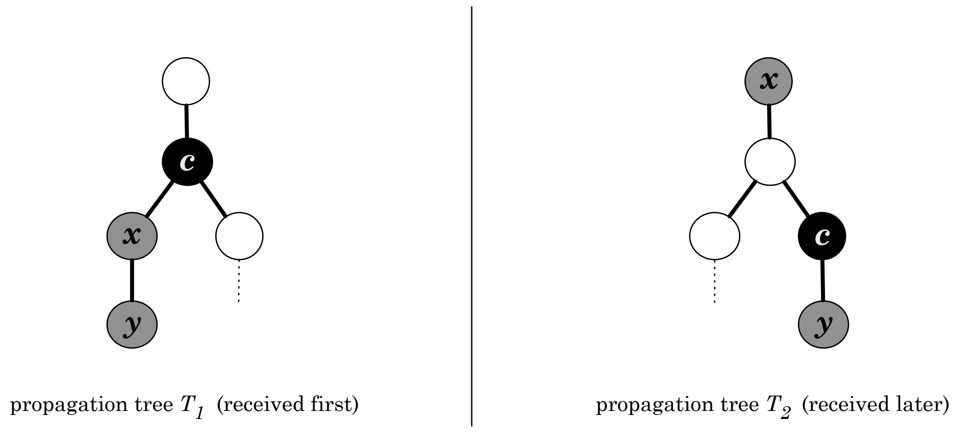

c
x
y
propagation tree T_1 (received first)
x
c
y
propagation tree T_2 (received later)

Figure 10: Two conflicting propagation trees received by c

below. The second case, formalized by Condition 3, is more complex and explained thanks to the example of Figure 10.

$$\exists\ y \in T_2.children(c) - T_1 \tag{2}$$

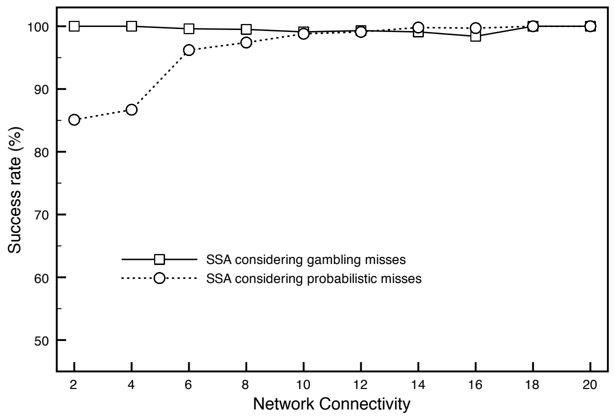
$$\begin{gathered}\exists\ y \in (T_2.children(c) \cap T_1)) \wedge \\ (x \in T_2 \mid c \in T_2.children(x) \wedge (y \in T_1.children(x)))\end{gathered} \tag{3}$$

In Figure 10, node c first receives tree T_1 and later T_2. When receiving T_2 from node x, c detects that node y might not be reached. Indeed, in both trees T_1 and T_2, y is a descendant of c. However, x is a descendant of c in T_1 and an ancestor of c in T_2. So, when c receives T_2 from x, it deduces that x was either not reached in T_1 or reached but decided to re-transmit though T_2. In both cases, c should retransmit, hence exceeds it quota, in order to reach y or in order to reach the node for which x decided to retransmit.

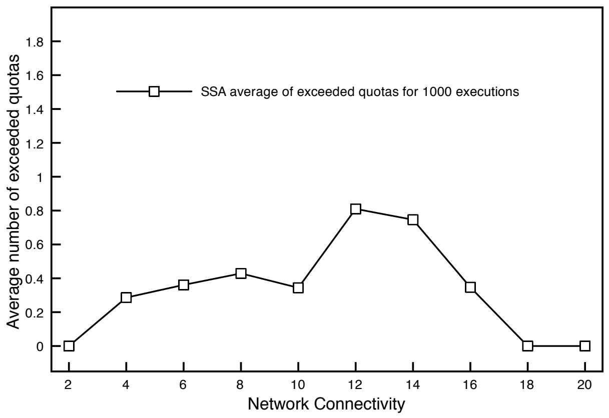
Countermeasure evaluation. When evaluating the effectiveness of our solution to mitigate gambling costs, we compared the final success rate of experiments implementing the proposed countermeasure with experiments that do not. In doing so, we varied the network connectivity c, while fixing the incrementing rate r to 2, the scope d defining the known subgraph of each process to 5, the range of loss probability L_i to $[0.05, 0.55]$, and the quota of messages q_i to 5.

Figure 11 (a) shows the success rate of executions implementing our countermeasure by varying the network connectivity, while Figure 11 (b) shows the corresponding average number of exceeded quotas based on 1000 distinct executions. When comparing curves of Figure 11 (a) and Figure 8 that shows the success rate considering the gambling misses,[5] we can see that our countermeasure significantly improves the final result. Furthermore, as shown in Figure 11 (b), the average number of exceeded quotas is negligible, i.e., less than one for 1000 distinct executions.

[5]Executions corresponding to these two curves have the same parameters.



(a) success rate



(b) average number of exceeded quotas

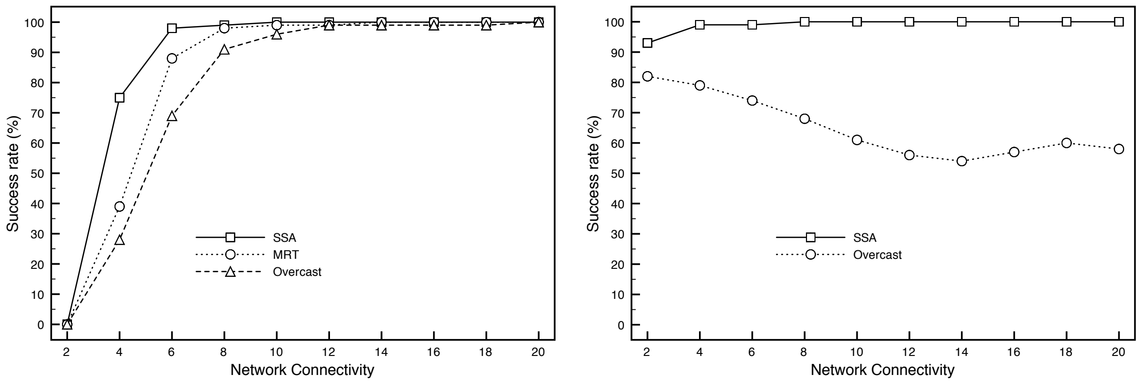
Figure 11: SSA with countermeasure, $Li \in [0.05, 0.55]$ and $q_i = 5$ messages.

4.3 Benefits of combined adaptiveness.

This section discusses the advantage of combining both resource and unreliability awareness when building the propagation tree, that is, it shows the benefit of our MPT construction technique. We compare our MPT to two relevant solutions. The first solution is inspired by the tree defined in Overcast [12]. Overcast is targeted at bandwidth-intensive applications. It defines a tree overlay that aims to maximize the bandwidth by placing nodes as far as possible from the root (the source) without sacrificing bandwidth. The available bandwidth resource in Overcast is modeled as weights assigned to links. In order to adapt the Overcast tree construction technique to our model, we consider the link weight as the number of messages assigned to the link, calculated by dividing the node quota by the number of its outgoing links in the tree. Thus, when building the Overcast tree in our model, at each iteration we add the link through which we can assign the maximum number of messages.

The second protocol is part of our previous work defined in [1] defining a reliable broadcast taking into account nodes failures probabilities (P_i) and links message loss probabilities (L_i). This broadcast solution is also based on a tree overlay named the *Maximum Reliability Tree* (MRT). This tree defines the most reliable tree of a known subgraph through which a message will be propagated. To avoid compromising this protocol, we assume for this comparison that each node will be able to send at least one message to each of its children in a tree. Thus, as shown in Figure 12 (a), each node quota of messages is equal to the number of direct neighbors, i.e., $\forall q_i$, $q_i = c$ the network connectivity.

When it comes to the limited knowledge each node has about the system, we assume that, in both our strategy and the compared protocols, nodes has only a partial view. Based on this knowledge, we use our Gambling increment strategy in order to build a propagation graph covering the whole system while using the different tree build criteria. Then, we apply our countermeasure to mitigate the *gambling misses* and focus our comparison in *probabilistic misses.* For fair comparison, we also apply our *Optimize()* function both to the Overcast tree and the MRT. Thus, once our comparison trees are built all nodes quotas are distributed in way to maximize the advantage of this resource.



(a) q_i= c

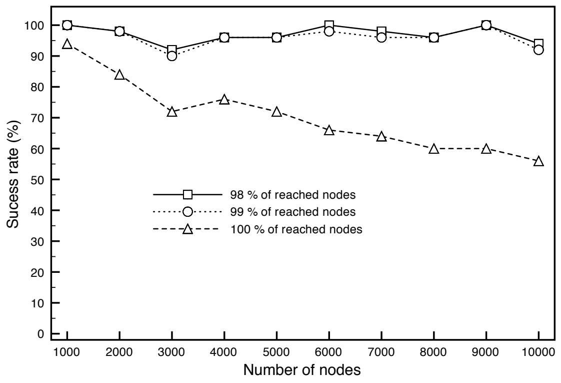
(b) q_i= 5

Figure 12: SSA reliability and bandwidth adaptiveness advantage

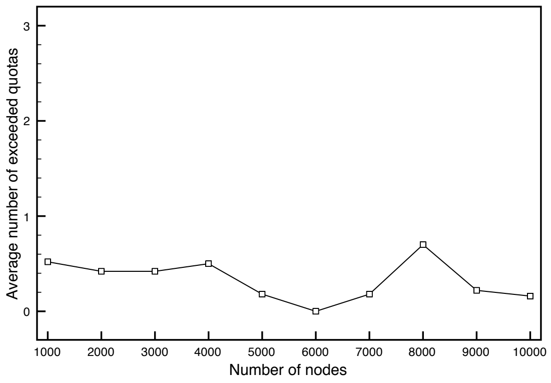
In this comparison, we vary the network connectivity c, while fixing the incrementing rate r to 2, the scope d defining the known subgraph of each process to 5 and the range of loss probability L_i to $[0.05, 0.55]$. As shown in Figure 12, the success rate of our approach is higher when using MPT than when using the Overcast tree and the MRT. When the quota is equal to the network connectivity c (Figure 12 (a)), the success rate of our approach and its comparison protocols increases as the c increases and thus as q_i increases. This reflects the capacity of available resources to hide the environment unreliability. When the quota q_i is fixed to 5 messages (Figure 12 (b)), our approach provides a higher reliability when using MPT than when using the Overcast tree. In addition, our approach has a different behavior than when using the Overcast tree while varying the network connectivity. Indeed, as the connectivity increases more links in the system are created offering a larger choice of links to the MPT construction technique. While the MPT takes advantage to include a more reliable links, the Overcast tree moves away from the line structure which imposes more leaves and thus more lost quotas. These latter quotas would contribute to hide the environment unreliability if not lost.

4.4 Scalability

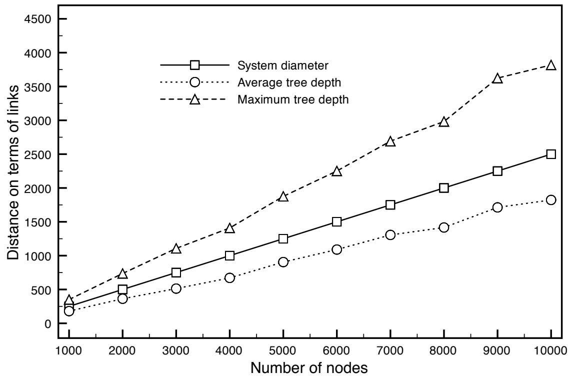
In order to evaluate the scalability of our algorithm, we performed several experiments with our simulation model, by drastically augmenting the number of processes in the system. In doing so, we considered all links to have the same loss probability $L \simeq 0.05$, and we fixed the scope d to 50, the incrementing rate r to 40 and the network connectivity c to 4. We also considered all individual quotas q_i to be the same and equal to the network connectivity, i.e. $\forall q_i$, q_i=4. Our scalability evaluation is pictured in Figure 13 (a) and Figure 14 (a) which show the rate of executions that succeeded to reach all nodes (100% of nodes), 99 % of nodes and 98% of nodes. In Figure 13 (a) the number of nodes in the system is varied in a linear way while in Figure 14 (a) it is varied in an exponential way. Figure 13 (b) and Figure 14 (b) then show the corresponding countermeasure price, in terms of the average of exceeded quotas needed to handle detectable gambling misses. Based on these Figures, we can conclude that our strategy provides a scalable streaming solution, with a graceful linear decrease as the number of processes in the system increases. We also notice that our solution requires a very small number of exceeded quotas to



(a) Success rate



(b) Average number of exceeded quotas



(c) System diameter and built tree depth

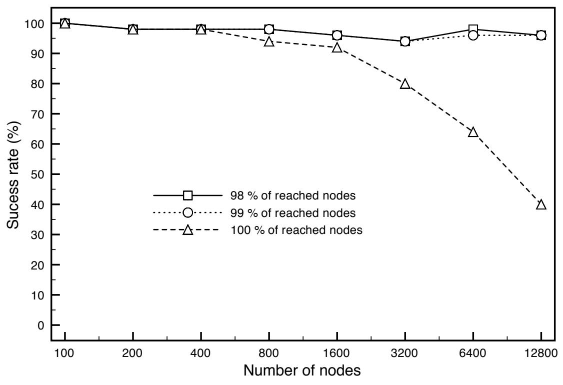
Figure 13: Scalability of SSA with linear growth of nodes, $L \simeq 0.05$ and $q_i = 4$ messages

correct cyclic conflicts.

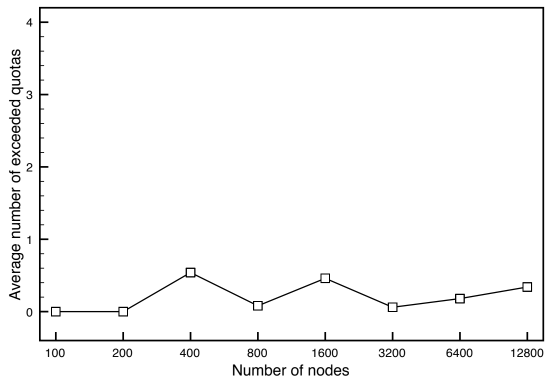
In each execution we have measured (1) the system diameter, computed as the number of links in the shortest path separating the most distant nodes; (2) the average tree depth, which represents the average distance, in terms of number of links, separating the source node to all the leaves in the resulting propagation graph; and (3) the tree depth in the propagation graph (i.e., maximum distance between the source node and the leaves). These measurements are shown in Figure 13 (c) and Figure 14 (c). Notice that the average tree depth is lower than the system diameter. This shows that while our tree construction technique aims at using the maximum of available resources, the resulting propagation graph is not a line, although a line is the topology that maximizes the use of quotas. Indeed, when enough quotas are available at some nodes (e.g., at hubs) our MPT construction algorithm assigns more than one children to those nodes, making the global tree shorter.

5 Related Work

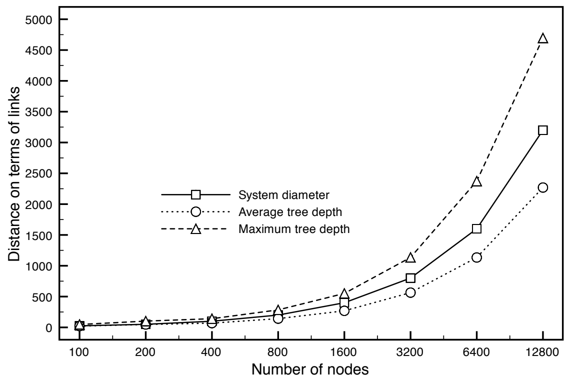
Several peer-to-peer streaming solutions have been proposed recently. Mainly, we can classify them into two classes: structured [4, 5, 7, 8, 12, 17, 18, 26] and unstructured [14, 19, 20, 21,



(a) Success rate



(b) Average number of exceeded quotas



(c) System diameter and built tree depth

Figure 14: Scalability of SSA with exponential growth of nodes, $L \simeq 0.05$ and $q_i = 4$ messages

23, 24, 29, 31]. The unstructured approach usually relies on a gossiping protocol, which consists in having each peer forward the data it receives to a set of randomly chosen neighbors. As a consequence, the path followed by the disseminated data is not deterministic. By contrast, the structured approach consists in first organizing the network peers into some overlay network and in routing disseminated data through this virtual topology.

These two approaches focus on different goals. Initially the structured sttrategy was devised to adapt to the underlying network characteristics, whereas the unstructured strategy, known as network agnostic, was devised for scalability. To ensure the same reliability, a structured dissemination uses fewer messages than an unstructured one. It however assumes that nodes have some knowledge about the network and imposes a computation overhead, which hinders scalability of these approaches.

Recently several researches worked to reduce the gap between the structured and the unstructured approaches. In the unstructured side, several approaches proposes a more deterministic forward decision in order to adapt some environment constraints or to avoid wasting resources by sending a duplicated messages. Along this line, [14, 15, 16] propose a gossip based strategies to ensure either an optimal reliability or an optimal delay by tuning the forward decision based on information about the neighbors received packets. That is, in order to reach some delay or rate targets, each node tries to answer the following question: which stream packets should be forwarded to which neighbor? Similar to our approach, [15] adresses the network links capacity limitations, however it does not consider these components unreliability.

In the structured side, several approaches propose an overlay construction mechanism to approach the scalability of the unstructured strategies. Our solution is part of this category. Along this line, several solutions are based on a tree have been proposed in the literature [7, 8, 12, 30]. Some of them define a multicast tree that aims at optimizing the bandwidth use [7, 6, 12]. Others, also deal with scalability by limiting the knowledge each process has about the system [8, 30]. Yet, other systems aim at increasing robustness with respect to packet loss [10, 11, 13]. Our approach differs from these systems in that it targets the three goals simultaneously. Our propagation structure is build collaboratively by distributed processes using their respective partial views of system. Reliability is accounted for by each process when building its local tree. Finally, bandwidth constraints are considered when defining how to forward packets along the propagation graph.

Narada [7] builds an adaptive *mesh* that includes group members with low degrees and with the shortest path delay between any pair of members. A standard routing protocol then is run on the overlay mesh. This work differs from ours by considering latency as the main cost related to links. While using the probing to change links in order to optimize the mesh, Narada does not take into account the loss probability of added or retrieved links. Furthermore, Narada nodes maintain a global knowledge about all group participants. In comparison, we take process and link failure probabilities into account and maintain local information only.

Regarding the forwarding load distribution, the work closest related to ours is Overcast [12], which leads to deep distribution trees. Such a tree would be our MPT in reliable environments, that is, if links do not lose messages.

Reducing the number of gossip messages exchanged between processes by taking the network topology into account is discussed in [27] and [28]. Processes communicate according to a predetermined graph with minimal connectivity to attain a desired level of reliability. Similarly to our approach, the idea is to define a directed spanning tree on the processes. Differently from ours, process and link reliabilities are not taken into account to build such trees.

Our strategy shares some design goals with broadcast protocols such as [1]. Both rely on the definition of a criteria for selecting the multicasting graph. In our strategy, however, we strive to both decrease packet loss and balance the forwarding load. The notion of *reachability probability* of a tree is presented in [1] to define the *Maximum Reliability Tree* (MRT). In our work, we define the reachabiliy probability of the streaming differently, by considering local knowledge only. These approaches illustrate a tradeoff in stream diffusion algorithms: while the protocol in [1] can lead to the optimum propagation tree, it requires global topology knowledge; our current algorithm relies on local knowledge but may not result in the optimal propagation tree.

When it comes to dealing with loops, which naturally appear in decentralized tree-based streaming solutions, several streaming solutions propose tree computation techniques that consist in dividing multicast members into groups. In such approaches, each group has a leader who is responsible of organising group members in a subtree, while leaders are in turn also organized in a tree [3, 5]. While this strategy prevent loops in the resulting overlay, it however penalizes the efficiency since all optimization are done locally to each group, i.e., nodes in different groups are unable to form overlay links.

Another set of tree-based solutions avoid loop problems by taking advantage of logical adress techniques, traditionally dedicated to routing solutions, in order to build a tree overlay. An example of such solutions is SplitStream [8], which builds several trees based on Scribe [4] and Pastry [9]. This approach ensures scalability since no computation is needed to define the tree. The routing is done implicitly by following the logical adresses assigned to members. The drawback of this approach is the absence of match between the overlay and the underlying physical network. That is, no efficiency guarantee can be ensured with this approach.

Our approach to detect loops, when building efficient tree overlays, differs from previous ones in that it ensures a resulting global tree close to the one built in a centralized manner, i.e., the tree we would obtain if we had a global knowledge about the system. In [25], we presented an overview of our solution focusing on the tree build technique, while providing no detail on our loop detection mechanism nor on its handling.

6 Conclusion

This paper introduces a probabilistic algorithm for reliable stream diffusion in unreliable and constrained environments. Differently from more traditional approaches, we resort to a "gambling approach," which deliberately penalizes a few consumers in rare cases, in order to benefit most consumers in common cases. Experimental evaluation has shown that our protocol outperforms gossip-based algorithms when subject to similar environment constraints. We believe that this main open up new directions for future work on large-scale data dissemination protocols.

References

[1] Garbinato, B. and Pedone, F. and Schmidt, R. (2004) An Adaptive Algorithm for Efficient Message Diffusion in Unreliable Environments. *Proceedings of DSN 04, Florence, Italy*, 507-516. IEEE

[2] Tanenbaum, A. S. (2002) Computer Networks. Prentice Hall PTR, NJ, USA.

[3] Chawathe, Y. and McCanne, S. and Brewer E. A. (2000) RMX: Reliable Multicast for Heterogeneous Networks. *Proceedings of INFOCOM 00, Tel Aviv, Israel*, 795-804. IEEE

[4] Castro, M. and Druschel, P. and Kermarrec, A. M. and Rowstron, A. I. T. (2002) Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE J. of Selected Areas in Communications.*, **20 (8)**, 1489-1499.

[5] Banerjee, S. and Bhattacharjee, B. and Kommareddy, C. (2000) Scalable application layer multicast. *Proceedings of SIGCOMM 02, New York, NY, USA*, 205-217.

[6] Zheng, X. and Cho, C. and Xia, Y. (2008) Optimal Peer-to-Peer Technique for Massive Content Distribution. *Proceedings of INFOCOM 08, Phoenix, AZ, USA*, 151-155. IEEE

[7] Chu, Y. and Rao, S. and Zhang, H. (2000) A Case For End System Multicast. *Proceedings of SIGMETRICS 00, Santa Clara, CA, USA*, 1-12. ACM

[8] Castro, M. and Druschel, P. and Kermarrec, A.M. and Nandi, A. and Rowstron, A. and Singh, A. (2003) SplitStream: High-Bandwidth Multicast in Cooperative Environments. *Proceedings of SOP 03, Bolton Landing, NY, USA*, 298-313. ACM

[9] Rowstron, A. and Druschel, P. (2001) Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proceedings of Middleware 01, Heidelberg, Germany*, 329-350. Springer-Verlag, Berlin.

[10] Apostolopoulos, J. G. (2001) Reliable video communication over lossy packet networks using multiple state encoding and path diversity. *Proceedings of VCIP 01, San Jose, CA, USA*, 392-409. SPIE

[11] Apostolopoulos, J. G. and Wee, S. J. (2001) Unbalanced multiple description video communication using path diversity. *Proceedings of ICIP 01, Thessaloniki, Greece*, 966-969. IEEE

[12] Jannotti, J. and Gifford, D. K. and Johnson, K. L. and Kaashoek M. F. and O'Toole, Jr, J. W. (2000) Overcast: Reliable Multicasting with an Overlay Network. *Proceedings of OSDI 00, San Diego, CA, USA*, 197-212. USENIX

[13] Nguyen, T. and Zakhor, A. (2002) Distributed video streaming with forward error correction. *In International Packet Video Workshop 02, Pittsburgh PA, USA.*.

[14] Bonald, T. and Massoulié, L. and Mathieu, F. and Perino, D. and Twigg, A. (2008) Epidemic live streaming: optimal performance trade-offs. *Proceedings of SIGMETRICS 08, Annapolis, MD, USA*, 325-336. ACM.

[15] Massoulié, L. and Twigg, A. and Gkantsidis, C and Rodriguez, P (2007) Randomized Decentralized Broadcasting Algorithms. *Proceedings of INFOCOM 07, Anchorage, AK, USA*, 1073-1081. IEEE

[16] Sanghavi, S. and Hajek, B. and Massoulié, L. (2007) Gossiping with Multiple Messages. *Proceedings of INFOCOM 07, Anchorage, AK, USA*, 2135-2143. IEEE

[17] Francis, P. (2000) Yoid: Extending the Internet Multicast Architecture. AT&T Center for Internet Research at ICSI (ACIRI).

[18] Mathy, L. and Canonico, R. and Hutchison, D. (2001) An Overlay Tree Building Control Protocol. *Proceedings of NGC 01, London, UK*, 76-87. Springer-Verlag

[19] TVants *http://tvants.en.softonic.com/*.

[20] Sopcast *http://www.sopcast.com/*.

[21] UUsee *http://www.uusee.com/*.

[22] Coolstreaming *http://www.coolstreaming.us*.

[23] Hei, X. and Liang, C. and Liang, J. and Liu, Y. and Ross, K. W. (2006) Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System. *Proceedings of IPTV Workshop 06, Edinburgh, UK.*

[24] Zhang, X. and Liu, J. and Li, B. and Yum, Y. S. P. (2005) CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. *Proceedings of INFOCOM 05, Miami, FL, USA*, 2102-2111. IEEE

[25] Allani, M. and Garbinato, B. and Pedone, F. and Stamenkovic, M. (2007) Scalable and Reliable Stream Diffusion: A Gambling Resource-Aware Approach. *Proceedings of SRDS 07, Beijing, CHINA*, 288-300. IEEE

[26] Tran, D. A. and Hua, K. A. and Do, T. (2003) ZIGZAG: an efficient peer-to-peer scheme for media streaming. *Proceedings of INFOCOM 03, San Franciso, CA, USA*, 1283-1292. IEEE

[27] Lin, M.-J. and Marzullo, K. (1999) Directional Gossip: Gossip in a Wide Area Network. University of California, San Diego, CA, USA.

[28] Lin, M.-J. and Marzullo, K. and Masini, S. (1990) Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks. University of California, San Diego, CA, USA.

[29] Kermarrec, A. M. and Massoulie, L. and Ganesh, A.J. (2001) Probabilistic reliable dissemination in large-scale systems.Microsoft Research, Cambridge, UK.

[30] Kostic, D. and Rodriguez, A. and Albrecht, J. and Bhirud, A. and Vahdat, A. (2003) Using Random Subsets to build Scalable Network Services. *USITS 03, Seattle, WA, USA*. USENIX

[31] Eugster, P. and Guerraoui, R. and Handurukande, S. and Kermarrec, A.-M. and Kouznetsov, P. (2001) Lightweight probabilistic broadcast. *Proceedings of DSN 01, Gteborg, Sweden*, 443-452. IEEE

[32] Aho, A. V. and Hopcroft, J. E. and Ullman, J. (1987) Data structures and algorithms. Addison Wesley.