A Gambling Approach to Scalable Resource-Aware Streaming*

Mouna Allani† Benoît Garbinato†
Fernando Pedone‡ Marija Stamenković‡

†University of Lausanne, CH-1015 Lausanne, Switzerland
E-mail: {mouna.allani,benoit.garbinato}@unil.ch

‡University of Lugano (USI), CH-6900 Lugano, Switzerland
E-mail: {fernando.pedone,marija.stamenkovic}@lu.unisi.ch

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.*

Keywords: *large-scale systems, reliable streaming, resource awareness.*

1 Introduction

Reliable stream diffusion under constrained environment conditions is a fundamental problem in large-scale distributed computing. Many internet systems (e.g., peer-to-peer, collaborative applications) rely on streaming multicast; consequently, their performace depends on the performance of the underlying streaming mechanism. Environment conditions are 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 nor 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 diffuse streams 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 max-

*This research is partially funded by the Swiss National Science Foundation, in the context of Project number 200021-108191.

imum probability of reaching all processes in its subgraph when subject to bandwidth constraints. MPTs are composed in a manner that respects bandwidth constraints.

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 memory constraints.

Besides discussing a new reliable stream diffusion algorithm, we show that it can be implemented in a very modular way, lending itself to real deployment. Briefly, our solution consists in decomposing the problem of reliable stream diffusion into four sub-problems. This decomposition of concerns gives rise to an architecture composed of four layers: The top layer, *Scalable Streaming Algorithm (SSA)*, is responsible for breaking the outgoing stream into a sequence of messages on the producer side and assembling these messages back into an incoming stream on the consumer side. Stream routing is encapsulated in an inner layer, the *Packet Routing Algorithm (PRA)*; messages are forwarded according to the global propagation graph, covering the whole system. Propagation trees are built out of MPTs using the *Propagation Tree Algorithm (PTA)* layer, which in turn relies on the partial view delivered by the *Environment Modeling (EML)* layer. This latter layer allows nodes to probe their neighborhood, determine their local membership, and assess the reliability of their neighbors and the links connecting them. Local knowledge is approximated using Bayesian inference.

Summing up, this paper makes the following contributions: (1) it introduces a novel algorithm for reliable message diffusion, which outperforms current solutions under constrained circumstances; (2) it shows how it can be engineered into a real system using a modular approach; and (3) it evaluates its performance and compares it to traditional gossip-based algorithms.

Roadmap. 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-constrained environment, the main challenge then consists in routing stream messages in a way that efficiently uses available resources. As already mentioned, memory and bandwidth are the resources that we consider in this paper.

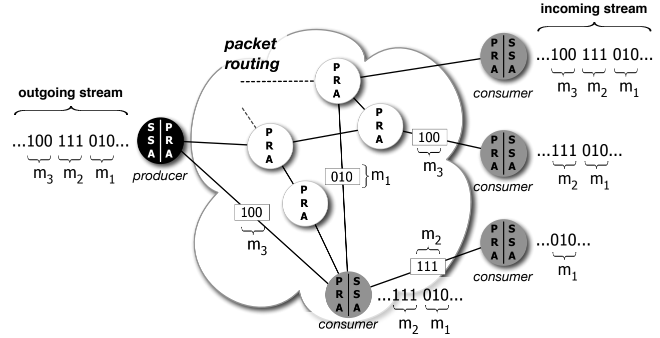


Figure 1. Stream diffusion scenario

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 n processes and $\Lambda = \{l_1, l_2, ...\} \subseteq \Pi \times \Pi$ is a set of bidirectional communication links. That is, we have $V(G) = \Pi$ and $E(G) = \Lambda$. Finally, 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 as 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 the limited bandwidth and memory available to each process?*

Formally, the *limited bandwidth constraint* is modeled as $Q = (q_1, q_2, ..., q_n)$, the set of quotas associated to processes in the system, i.e., q_i is the *individual quota of messages* at disposal of process p_i to forward a single stream

packet. 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 the *limited memory constraint*, 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. That is, 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 [8], 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 [1]. 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 [8], while taking into account strict constraints on bandwidth and on memory. 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.

From resource constraints to gambling. 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 yes.

Intuitively, our approach consists in having processes make bold decisions, in spite of their limited view on the system (scope), in the hope to better use the available bandwidth (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 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 four-layer architecture pictured in Figures 3 and 4. The top layer represents a standard stream fragmentation layer. It executes the *Scalable Streaming Algorithm (SSA)*, which 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. 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 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 Figures 2, 3 and 4, 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

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

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 goes beyond the scope of this paper and can be found in [8]. Finally, the *Unreliable Link Layer (ULL)* allows each process p_i to send messages to its direct neighbors in a probabilistically unreliable manner.

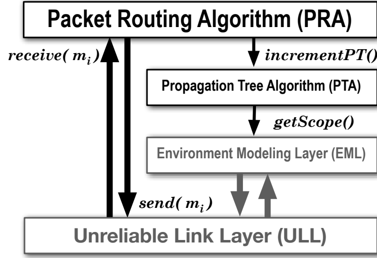


Figure 2. Router node architecture

3.3 Scalable Streaming Algorithm

The scalable streaming solution, presented in Algorithm 1, is fairly straightforward. On the producer side, as long as 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 lost or out-of-sequence packets when streaming realtime data, such as audio or video streams.

3.4 Packet Routing Algorithm

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 propagation graph itself might well not be a tree.

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: $lastSeq \leftarrow m.seq$

Algorithm 1. Scalable Streaming Algorithm (SSA) executed by 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$(p_k, p_i) = r$ **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

Algorithm 2. Packet Routing Algorithm (PRA) executed by p_i

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 3.6. At this point, all

we need to know is that it 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 the 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.

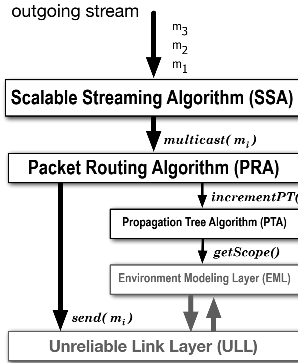


Figure 3. Producer node architecture

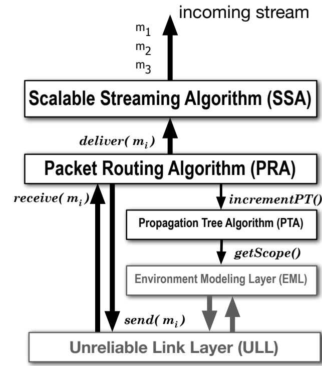


Figure 4. Consumer node architecture

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

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).[2] 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 3.6. For now, all we need to know is that an 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.

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** $pt \cup lpt_i$

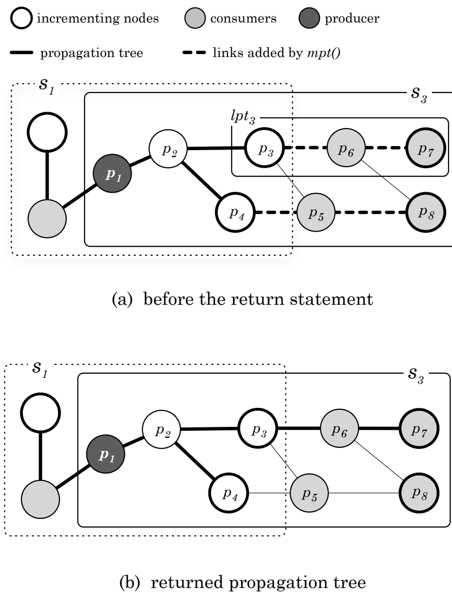
Algorithm 3. Propagation Tree Algorithm (PTA) executed by p_i

The gambling effect. Intuitively, the approach taken by the $mpt()$ primitive consists in augmenting pt with the best

[2] The conditional nature of this increment is motivated by performance concerns: during stable periods of the system, propagation trees remain unchanged, cutting down the processing load of incrementing nodes.

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 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 5 illustrates the propagation tree increment process 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 5 (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 5 (a) 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 5 (b) pictures the new propagation tree resulting from the above increment process.



(a) before the return statement

(b) returned propagation tree

Figure 5. Propagation tree increment

3.6 Maximum Probability Tree

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 need to introduce the notions of *reachability probability* and *reachability function*. These notions are borrowed from [8].

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 [8]. That is, this version assumes that only links can fail by losing messages with a given probability, whereas processes are assumed to be reliable.[3]

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

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 18). For this, function $optimize()$ allots messages of q_s one by one, until all messages have been allocated (line 16 to 18). 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 17). 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

[3]Note that this simplification causes no loss of generality; see [8] for details.

1: **function** $mpt(S, T)$
2: **while** $V(S) - V(T) \neq \emptyset$ **do**
3: $O \leftarrow \{l_{j,k} \mid l_{j,k} \in E(S) \wedge p_j \in V(T) \wedge p_k \in V(S) - V(T)\}$
4: **let** $l_{u,v} \in O$ **such that** $\forall l_{r,s} \in O:$
5: $maxR(T \cup p_v) \geq maxR(T \cup p_s)$
6: $T \leftarrow T \cup \{p_v\}$
7: **return** T

8: **function** $maxR(T)$
9: $\vec{m} \leftarrow optimize(T)$
10: **return** $R(T, \vec{m})$

11: **function** $optimize(T)$
12: **let** $\vec{m}$: $\forall l_j \in E(T)$, $\vec{m}[j]$ **is the number of messages to be sent through link** l_j
13: $\vec{m} \leftarrow (0, 0, \cdots, 0)$
14: **for all** $p_s \in V(T)$ **do**
15: **let** $\Lambda_s \subset E(T):\ l_k \in \Lambda_s \Rightarrow (p_s, p_k) \in E(T)$
16: **while** $\sum_{l_k \in \Lambda_s} \vec{m}[k] < q_s$ **do**
17: **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**
18: $\vec{m} \leftarrow \vec{m}_u$
19: **return** $\vec{m}$

Algorithm 4. MPT Building Process

$mpt()$, 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 6 to 8 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., $\forall p_i : q_i = 3$.

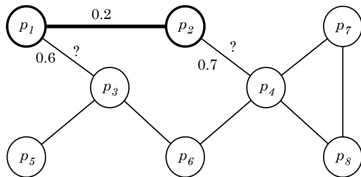


Figure 6. Resulting tree after the first iteration step

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 7 as trees T' and T'' respectively.

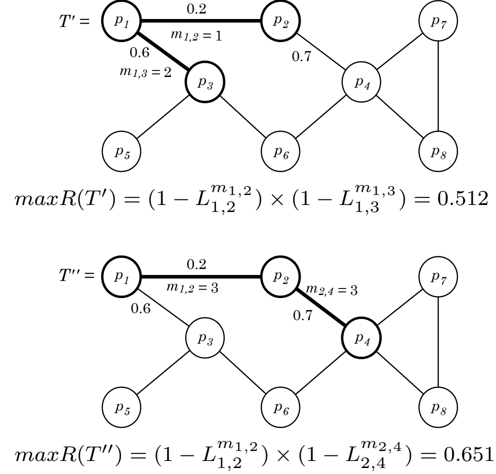


Figure 7. Alternative trees during the second iteration step

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 adding link $l_{2,4}$ rather than link $l_{1,3}$, although the latter is more reliable. Figure 8 pictures the final Maximum Probability Tree returned by function $mpt()$.

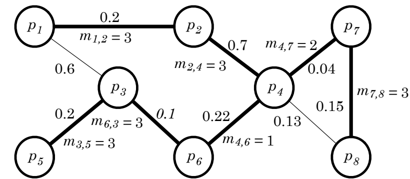


Figure 8. Final Maximum Probability Tree

4 Performance Evaluation and Discussion

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 3.6, this does not compromise the generality of our approach. We performed experiments with 100 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. 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 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 as a success when the diffused stream packet reaches all nodes in the system.

4.1 Benefits of gambling

To measure the benefit of our gambling approach, we compare our *Scalable Streaming Algorithm* (SSA) with a modified version of *Bimodal Multicast Algorithm* (BMA), proposed in [4]. Our modified version of BMA implements the notion of individual quota. That is, 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[4] 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 9 shows the evolution of the success rate of SSA and BMA respectively, when varying the network connectivity. As we can see, the success rate of BMA 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 focuses its efforts on less reliable paths. That is, as the connectivity increases, 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. More precisely, 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 of children of each process to its quota. As shown in Figure 9, 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

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

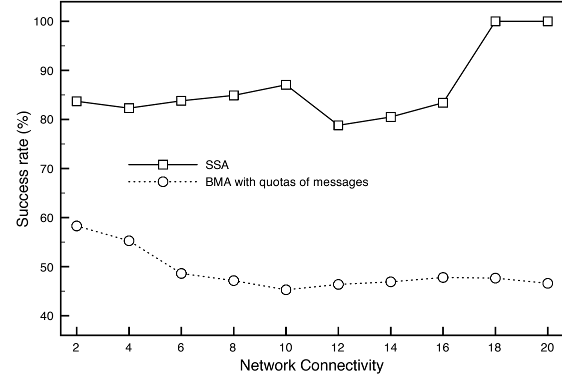


Figure 9. SSA *vs.* BMA with quotas

connectivities between 10 and 16. As discussed hereafter, this drop constitues the costs of gambling.

4.2 Cost of gambling

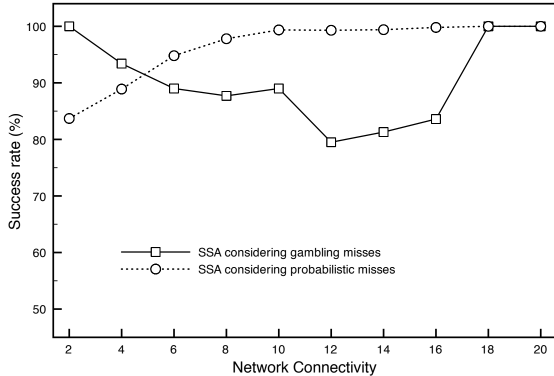
To evaluate our gambling approach, we measured the success rate by considering the two types of missed[5] executions: the probabilistic misses and the gambling misses (Figure 10). The *probablistic misses* are due to losing messages sent through unreliable links. These misses simply come from the probabilistic model we consider. Gambling misses are due to executions in which the *effective* propagation graph[6] does not cover the whole system.

In Figure 10 (a), we see that the connectivity variations have different impacts on probabilistic and gambling misses. Considering probabilistic misses, we note that as the connectivity increases, the probability of reaching all nodes increases. Indeed, 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, as the connectivity increases, misses due to the structure of the *effective propagation graph* become more frequent since there is a larger choice of links, which induces a higher risk to make contradictory decisions when building distinct *propagation trees*. However, when reaching high connectivity (12 in our example), this type of misses becomes less frequent since the known scope of each process becomes close to the whole system.[7] Many cases of gambling misses are detectable and could

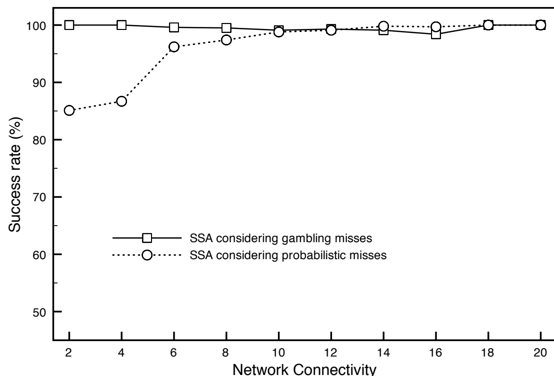
[5]A *miss* is an execution where some nodes in the system never receive the diffused packet.

[6]An effective propagation graph results from the aggregation of effectively followed *propagation trees*.

[7]When the known 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 risk.



(a) without countermeasures



(b) with countermeasures

Figure 10. Gambling cost on the success rate

be resolved via a simple countermeasure, which implies to slightly exceed some individual quotas. Due to lack of space, we do not describe the details of these countermeasures here. Figure 10 (b) however presents the results of these countermeasures on the reachability probability ensured by SSA.

5 Related Work

Several application-level multicast systems based on a tree have been proposed in the literature [5, 6, 9, 11]. Some of them define a multicast tree that aims at optimizing the bandwidth use, notably Narada [6] and Overcast [9]. Others, also deal with scalability by limiting the knowledge each process has about the system [5, 11]. Yet, other systems aim at increasing robustness with respect to packet loss [2, 3, 14]. 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 [6] 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 probably Overcast [9], which leads to deep distribution trees. Such a tree would be our MPT in reliable environments, that is, if links do not lose messages.

In [7] and [10] the authors show how to implement a gossip-based reliable broadcast protocol in an environment in which processes have a partial view of the system membership. Our protocols as well do not require processes to know all the system members or the topology connecting them. In addition to [7] and [10], our approach takes reliability properties of processes and links into account in order to ensure reliable broadcast.

Reducing the number of gossip messages exchanged between processes by taking the network topology into account is discussed in [12] and [13]. Processes communicate according to a pre-determined 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.

Finally, our strategy shares some design goals with broadcast protocols such as [8]. 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 [8] to define the *Maximum Reliability Tree* (MRT). This tree defines the most reliable tree of a known subgraph through which a message will be propagated. In our work, we define the reachabiliy probability of the streaming differently, by considering local knowledge only.

The approaches illustrate a tradeoff in stream diffusion algorithms: while the protocol in [8] 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 optimum information propagation tree.

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. Our current work is investigating alternative gambling algorithms.

References

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison Wesley, 1987.

[2] J. G. Apostolopoulos. Reliable video communication over lossy packet networks using multiple state encoding and path diversity. In *Visual Communications and Image Processing*, January 2001.

[3] J. G. Apostolopoulos and S. J. Wee. Unbalanced multiple description video communication using path diversity. In *IEEE International Conference on Image Processing*, October 2001.

[4] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[5] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of ACM SOP*, October 2003.

[6] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics*, pages 1–12, June 2000.

[7] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 443–452, July 2001.

[8] B. Garbinato, F. Pedone, and R. Schmidt. An adaptive algorithm for efficient message diffusion in unreliable environments. In *Proceedings of IEEE DSN*, pages 507–516, June 2004.

[9] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI*, October 2000.

[10] A.-M. Kermarrec, L. Massoulie, and A. Ganesh. Probabilistic reliable dissemination in large-scale systems. Technical report, Microsoft Research, June 2001.

[11] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proceedings of USITS*, March 2003.

[12] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, June 1999.

[13] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small networks. Technical Report CS1999-0637, University of California, San Diego, Nov. 1999.

[14] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *Packet Video Workshop*, 2002.