Spring: Fountain-Coded Multicast as a Service

Xindan Zhang, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—Large language models are increasingly post-trained with reinforcement learning across geographically distributed datacenters. In each round, gigabytes of model weights need to be sent from the trainer to multiple inference workers — a classic multicast for which performance needs to be maximized. Despite over two decades of research on maximizing overlay multicast throughput, existing work focused on individual aspects of the solution space, such as tree construction, coding, or overlay transport. In this paper, we present *Spring*, a complete system design and implementation that incorporates state-of-the-art point solutions: a conceptual-flow planning target, explicit tree packing, RaptorQ as a fountain code, and a tailored fountain-coded overlay transport. *Spring* is implemented in Rust so that it is highly performant, yet is wrapped in a convenient Python API for RL workloads. In a Qwen3-8B transfer involving seven geographically distributed workers, *Spring* delivers a 16.39 GB update at 1212 Mbps, compared with 431 Mbps with Cloudcast, a recent cost-optimized solution.**

I. Introduction

With the ubiquity of large language models (LLMs), it has become imperative for large corporations and small enterprises alike to post-train these models with private data on leased GPU clusters. The prevailing post-training paradigm is reinforcement learning (RL), which has fueled much of the recent meteoric rise in LLM capabilities, especially in mathematics and coding [1]–[3].

Training an LLM using RL, however, is prohibitively costly in GPU compute: it utilizes synthetic rollout generators and iterative policy updates to improve a model's reasoning capabilities, which requires a large number of *inference workers* and a *trainer* to run in a collaborative fashion. In a typical RL training workload, the trainer periodically produces a new policy version, serialized into a model artifact that is broadcast to the inference workers over the network.

As it is less cost effective to lease a large amount of GPU compute in a single datacenter, it has been a recent industry trend — exemplified by Prime Intellect's INTELLECT-2 [4] — to run *geographically distributed* RL post-training workloads that span multiple regions, communicating with one another over a wide-area inter-datacenter network. As all inference workers run independently of each other once they have the same policy version, such RL post-training workloads are inherently parallelizable without an inter-worker communication bottleneck, unlike conventional pretraining or supervised fine-tuning workloads.

Despite such inherent parallelism between inference workers, the trainer still needs to *broadcast* its updated model weights to all inference workers. A large amount of data — easily in the order of hundreds of Gigabytes for modern LLMs — still needs to be broadcast between the trainer (as the sender) and the inference workers (as the receivers) [1], [5]. This broadcast sits on the critical path of on-policy workloads: inference workers cannot generate rollouts for a new policy version until its weights arrive, and at wide-area rates a multi-gigabyte update idles them for minutes per training round. Given known link capacities over the inter-datacenter network, the problem of maximizing the throughput between the trainer and inference workers is a classic problem of maximizing multicast throughput over overlay networks with link capacity constraints, which has been thoroughly studied in theory over the past two decades [6]–[9].

Unfortunately, implementing two decades' worth of solutions on overlay multicast into a working system for geo-distributed RL is far from straightforward. Existing pieces of related work typically provided point solutions in a multi-dimensional solution space: one line of work focused on tree construction, another on coding, and perhaps a third on transport protocols. As examples, conceptual-flow formulations reason about the maximum multicast throughput on an overlay, while tree-packing methods turn capacity into multiple multicast structures [6]–[10]. Further, fountain codes such as RaptorQ [11] provided the foundation for moving data across relays in a loss-resilient fashion.

We argue that it is not enough to know that the maximum achievable multicast rate can be characterized in theory, or that a good erasure code exists, or that a relay overlay can forward packets. We need an integrated, complete, and highly performant system design: from planning checkable multicast trees, to sending RL policy updates reliably and reacting to slow links. The formidable challenge is to build a real-world system that actually works well for real-world RL workloads.

In this paper, we present *Spring*, a multicast substrate as a service for geo-distributed RL post-training. *Spring* represents a new, end-to-end system implementation of state-of-the-art point solutions in Rust, built upon *Nextmini*, a general-purpose networking research framework [12]. As Fig. 1 shows, the trainer and inference workers access the service through a Python API. Internally, *Spring* separates control-plane planning from data-plane execution: the control plane coordinates directed probes among participating nodes and collects the reported rates in a capacity matrix. From this matrix, it obtains a multicast-rate upper bound from *conceptual flows* [9] and uses that bound as its planning target. It then searches for a receiver-covering tree family and checks that the returned family attains the target within the measured capacities.

979-8-3195-0662-7/26/$31.00 ©2026 IEEE

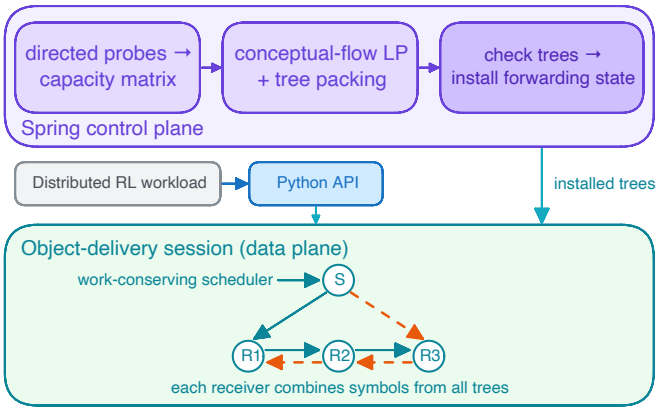


Fig. 1. An overview of *Spring*'s architecture. For clarity, only two of the installed trees are shown, solid and dashed: each reaches every receiver, and receivers forward for one another.

To utilize the planned trees as much as possible, *Spring*'s data plane uses lossless overlay links over TCP to propagate any potential congestion upstream to the source using backpressure. A highlight is *Spring*'s reactive, backpressure-driven scheduler, which sends fountain-coded symbols — implemented with RaptorQ — to non-backlogged "writable" trees. A slow tree therefore delays only the symbols already sent to it, while non-backlogged trees keep advancing the receivers' accumulation of coded symbols. Each receiver combines the symbols delivered over all trees for a source block, and reconstructs the block as soon as that combined set is decodable. In essence, fountain coding is used for cross-tree collaboration, rather than erasure correction.

The original contributions in this paper are three-fold. *First*, we turn a measured directed overlay into a checked, installable family of receiver-covering trees. *Spring* uses the conceptual-flow optimum as a planning target, constructs a tree packing under the measured capacities, and checks the returned family's rate and edge loads before installation. *Second*, we bring RaptorQ's symbol interchangeability into multi-tree multicast in one end-to-end, real-world system: to exploit it fully, we develop reactive, backpressure-driven sending that runs the installed trees as equals, without enforcing per-tree planning rates or assigning a fixed portion of the object to a tree. *Finally*, we expose *Spring* to RL training workloads through a simple Python API. A training workload issues one lossless object transfer per policy update, without managing the underlying data movement itself. *Spring* is open-source under the AGPLv3 license at https://github.com/iqua/nextmini.

We evaluate *Spring* from planning to real-world delivery over wide-area networks (WANs). Across 200 role-constrained directed overlays, every case we test returns a set of trees whose rate matches the conceptual-flow optimum within numerical precision. Our single-machine experiments validate *Spring*'s runtime design, including a $4.07\times$ throughput gain over fixed striping when an installed tree slows. In a seven-receiver WAN transfer of a 3 GiB payload, *Spring* utilizes 14 receiver-covering trees and reaches 1488 Mbps of average throughput, a $4.3\times$ improvement over the best fixed-topology baseline. Finally, in a Qwen3-8B GRPO-style policy-weight transfer, *Spring* delivers a 16.39 GB update in 110.9 s on average over its installed trees, reaching 1212 Mbps, compared with 431 Mbps for Cloudcast [10].

II. Related Work

Four lines of work meet in *Spring*: distributed RL systems require a fast broadcast, conceptual flows bound how fast the overlay can make it, fountain codes decouple the delivered bytes from the paths that carry them, and overlay multicast systems supply the deployment craft.

Distributed RL. IMPALA [13] and Ape-X [14] popularize distributed actor-learner architectures that decouple rollout generation from policy optimization. Frameworks such as Ray RLlib [15] expose this separation through reusable policy-evaluation and policy-optimization abstractions. Recent LLM post-training systems push the same pattern toward larger policies and larger worker fleets [1], [5], making repeated policy dissemination an increasingly important systems cost. INTELLECT-2 [4], which runs RL post-training across globally decentralized resources, streams checkpoint shards through a CDN-like network of HTTP relay servers. *Spring* complements these systems by optimizing the actual multicast itself, over trees planned from measured link capacities.

Conceptual flows and network coding. Maximizing one-to-many throughput over a capacitated overlay is a classic multicast problem [6], [7]. Building on this line of work, within one multicast session, each receiver can be treated as receiving its own *conceptual unicast flow* from the source. These conceptual flows do not additively compete with each other on an edge; instead, the session reserves enough rate on each edge to carry the *maximum* conceptual rate traversing it. This observation yields a linear optimization formulation that jointly selects paths and rates [8], [9].

With coding permitted at every relay, the classical single-source multicast theorem makes the conceptual-flow bound attainable; coding can, but need not, improve throughput over replication-only routing [16]–[18]. Recoding large model artifacts inside the network is costly in practice, in relay compute, deployment, and debugging alike. *Spring* therefore keeps its relays replication-only and uses the conceptual-flow optimum as an upper bound. In Section IV-B, we give a sufficient capacity condition for source-side egress relays under which replication attains this bound; *Spring* checks the explicit tree family returned for every instance.

Fountain-style erasure coding over overlays. A fountain code turns an object into a potentially unbounded stream of coded symbols, allowing each receiver to reconstruct the object once it has collected a sufficiently large subset. The abstraction was introduced as the *digital fountain* [19], for one-to-many bulk delivery. Fountain coding neither requires nor substitutes for network coding at intermediate nodes, since a source-side code cannot raise the rate a session can achieve, only change which symbols that rate carries. Classical erasure codes such as Reed-Solomon instead fix the number of coded symbols in advance [20]. LT codes [21] provided the first full realization

of the fountain, and Raptor codes [22] then brought encoding and decoding down to linear time. RaptorQ, standardized for reliable object delivery [11], operates independently on source blocks and is systematic within each block: it sends source symbols unmodified and generates repair symbols as needed. Byers et al. carry the fountain code from IP multicast onto overlay networks [23]. *Spring* adopts RaptorQ for its low-complexity, systematic design and pairs it with bounded-queue backpressure, which preserves admitted symbols when a downstream queue fills. Although the standard calls additional encoded symbols repair symbols, *Spring* uses them to complete a source block while some of that block's source symbols are still in flight on a slower tree; we therefore call them *extra coded symbols*.

Wide-area and overlay multicast. Application-layer multicast (ALM) [24]–[28] emerges as a deployable alternative to IP multicast, organizing end systems or overlay servers into delivery trees, multi-tree forests, or meshes. Classic systems explore bandwidth efficiency, low delay, robustness, scalability, and adaptation to churn. At the infrastructure layer, B4 and SWAN [29], [30] centrally engineer traffic across provider-operated inter-datacenter WANs. A separate line targets point-to-multipoint bulk transfers: DCCast [31] uses one forwarding tree per transfer, whereas QuickCast and FlexCast [32], [33] partition receivers across multiple trees to improve completion time. CodedBulk [34] performs network coding at intermediate datacenters and uses hop-by-hop flow control to accommodate asymmetric link rates. Cloudcast [10] targets multi-cloud replication, minimizing egress and VM cost within a replication-time budget by selecting waypoint VMs and binding fixed-size stripes to multicast trees.

Spring organizes its user-space overlay through a logically centralized, SDN-style control plane. It uses the conceptual-flow optimum as a target and checks whether an explicit set of receiver-covering trees realizes it under the measured edge capacities. Unlike Cloudcast's fixed stripe-to-tree assignment, *Spring* schedules the next useful symbol on any currently writable installed tree. To enable collaboration across the installed trees, *Spring* performs RaptorQ encoding at the source: each receiver combines arriving symbols from all trees to decode each source block. CodedBulk recodes at intermediate datacenters; *Spring*'s intermediate nodes simply replicate symbols unchanged.

III. Problem Statement

Distributed RL separates policy optimization at a trainer from rollout generation across inference workers. When these workers span multiple datacenters, distributing policy weights becomes a recurring wide-area one-to-many transfer. For each policy version selected for dissemination, the trainer holds a B-byte object that every target inference worker reconstructs before loading that version. *Spring* treats each such dissemination as one exact-object multicast session.

Network model. A *Spring* session originates at a single source s, the trainer. The source datacenter may also host a possibly empty set $\mathcal{A}$ of *source-side egress relays*, colocated with the trainer and connected to it by high-capacity intra-datacenter links. For each $a \in \mathcal{A}$, this local connection $e_a = (s, a)$ is its *source-side egress-relay edge*. It allows the relay to receive one copy from the trainer and fan it out through its own wide-area connections. Across the wide area, the receiver set $\mathcal{R}$ consists of inference workers distributed among remote datacenters. Every receiver reconstructs the policy object; a receiver with downstream children also forwards arriving symbols to them.

With multi-hop multicast, a symbol received by an egress relay or an inference worker can be replicated to its downstream receivers, so the transfer distributes fanout across outgoing capacity at both the source site and the receiver sites, rather than consuming the trainer's own egress once per receiver.

We represent these nodes as a directed overlay $\mathcal{G} = (V, E)$ with $V = \{s\} \cup \mathcal{A} \cup \mathcal{R}$, where E is the set of unicast connections available to *Spring*. An edge $e = (u, v) \in E$ means that *Spring* can send data from u to v. Before planning, *Spring* probes each connection and records the measured rate as $c_e > 0$, the planning capacity of the corresponding edge.

The conceptual-flow formulation uses these edge capacities to compute a multicast-rate planning target for the full receiver set. Its edge-level solution leaves the parent–child relationships for packet replication unspecified, so an executable plan requires explicit forwarding structures. We therefore define a *receiver-covering tree* $T = (V_T, E_T)$ as an out-arborescence of $\mathcal{G}$ rooted at s that reaches every receiver in $\mathcal{R}$, using source-side egress relays from $\mathcal{A}$ when needed. Within a given tree, every node other than s has exactly one parent, and every branch ends at an inference worker. Egress relays may appear only as internal forwarding nodes, while inference workers may also forward to downstream children. Let $\mathcal{T}$ denote the set of all such trees, and let $P_T(s, d)$ denote the unique path from s to receiver d in tree T.

The planner therefore selects a finite, weighted collection of K receiver-covering trees, $\Lambda = \{(T_k, \lambda_k)\}_{k=1}^{K}$, where $T_k \in \mathcal{T}$ and $\lambda_k \geq 0$ is the rate allocated to that tree by the planning model. These values are used to account for edge capacity: tree T_k contributes load λ_k to every edge it contains, and the loads of trees sharing an edge add together.

We refer to Λ as the *tree family*. A tree family is feasible when its combined load stays within the planning capacity of every overlay edge:

$$\sum_{k:e \in E(T_k)} \lambda_k \leq c_e, \qquad \forall e \in E.$$

Because every selected tree reaches every receiver, a rate λ_k assigned to tree T_k is delivered to the entire receiver set. The planned multicast rate of the tree family is therefore $\rho(\Lambda) = \sum_{k=1}^{K} \lambda_k$. The planning problem is to maximize $\rho(\Lambda)$ subject to the edge-capacity constraints above.

Section IV-A and Section IV-B describe how *Spring* obtains a target rate from the conceptual-flow formulation of prior work, searches for an explicit receiver-covering tree family, and checks whether that family attains the target.

Object delivery. The tree family determines where data may travel, but not how a finite object is assigned to those

TABLE I
EXISTING FOUNDATIONS AND *Spring*'S SYSTEM CONTRIBUTIONS.

Layer	Existing foundation	*Spring* contribution
Planning	Conceptual flows, fractional tree packing, column generation, and Steiner pricing [8], [9], [35], [36]	A planner that constructs receiver-covering trees under measured edge capacities and checks whether the returned family attains the conceptual-flow target.
Object delivery	RaptorQ and coded overlay delivery [11], [23]	A backpressure-aware multi-tree runtime that sends the next useful symbol on any writable tree and combines symbols across trees at every receiver.
Service	Application-layer multicast [24], [28]	An end-to-end Rust service that integrates capacity probing, route installation, feedback-driven exact-object delivery, and a Python API for RL workloads.

trees. *Spring* keeps the installed family fixed for the session but does not bind a predetermined portion of the object to any tree or enforce the per-tree planning rates λ_k at runtime. For transmission, *Spring* divides the object into RaptorQ source blocks, whose source symbols carry the block's original bytes unchanged. As needed, the source generates extra coded symbols for the same block. RaptorQ encoding occurs only at the source, while block reconstruction occurs at the receivers. Within each block, any currently writable tree may carry the next source or extra coded symbol, and every receiver combines symbols delivered over all trees and reconstructs the block once that combined set is decodable. If a receiver or source-side egress relay forwards a symbol, it copies the symbol unchanged to its downstream children.

For a finite transfer over the installed trees, let τ be the measured delivery interval for a B-byte object. The observed throughput is $8B/\tau$ bits/s. Section IV-D describes how *Spring* implements this execution model.

IV. SYSTEM DESIGN

Fig. 1 provides an overview of *Spring*'s path from network measurement to policy-object delivery. The control plane uses the measured directed overlay to obtain a conceptual-flow rate target, construct and check a receiver-covering tree family, and install its forwarding state. The trainer then submits the policy object through *Spring*'s Python API. The data plane schedules RaptorQ symbols over the installed trees and uses receiver feedback until every receiver reconstructs the object. Table I separates the foundations that *Spring* adopts from its system contributions. The following subsections develop the design along this path.

A. Conceptual-Flow Target

Spring seeks a receiver-covering tree family of maximum planned multicast rate. Directly optimizing over all such trees is difficult because their number is exponential. It therefore adapts the conceptual-flow formulation of prior work to the measured directed overlay and uses the resulting optimum as its planning target [8], [9].

Let $\delta^+(v)$ and $\delta^-(v)$ denote the outgoing and incoming edges of node v. On *Spring*'s measured directed overlay, the conceptual-flow formulation uses edge-based variables. For every receiver $d \in \mathcal{R}$ and edge $e \in E$, let $f_d(e) \geq 0$ denote the conceptual flow on e toward d, and let $x_e \geq 0$ denote the shared conceptual load on that edge. The resulting linear program (LP) is

$$\max \quad \rho_{\mathrm{cf}} \tag{1}$$

$$\text{s.t.} \quad \sum_{e\in\delta^+(v)} f_d(e) - \sum_{e\in\delta^-(v)} f_d(e) = \begin{cases} \rho_{\mathrm{cf}}, & v = s, \\ -\rho_{\mathrm{cf}}, & v = d, \\ 0, & \text{otherwise}, \end{cases}$$
$$\forall d \in \mathcal{R},\ \forall v \in V, \tag{2}$$

$$0 \leq f_d(e) \leq x_e, \quad \forall d \in \mathcal{R},\ \forall e \in E, \tag{3}$$

$$f_d(e) = 0, \quad \forall d \in \mathcal{R},\ \forall e \in \delta^-(s) \cup \delta^+(d), \tag{4}$$

$$0 \leq x_e \leq c_e, \quad \forall e \in E. \tag{5}$$

The conservation constraints deliver the same rate ρ_{cf} from s to every receiver. Unlike independent unicasts, the receiver-specific flows do not add on a shared edge; x_e need only dominate each of them. This domination rule captures multicast sharing, while $x_e \leq c_e$ keeps the shared load within the measured capacity. The no-waste constraint keeps a flow for receiver d from entering the source or continuing beyond d. It does not prevent that receiver from forwarding symbols toward other receivers, whose conceptual flows are modeled separately.

The LP supplies the scalar target ρ_{cf}; its variables $f_d(e)$ and x_e establish that this target is feasible in the conceptual-flow relaxation, but do not prescribe the explicit replication trees. The tree-construction step therefore uses the same measured capacity vector c that constrained the LP.

All planning guarantees in this section are relative to this independent directed-edge capacity model. Section V separately measures the host and transport effects encountered during execution.

The conceptual-flow optimum is also an upper bound on any feasible family of receiver-covering replication trees.

Proposition 1 (Conceptual-flow relaxation bound)**.** *Every capacity-feasible packing of receiver-covering trees on $\mathcal{G}$ induces a feasible conceptual-flow solution of the same multicast rate. Therefore ρ_{cf} upper-bounds the rate of any such tree packing.*

Proof: For a feasible family Λ, define

$$f_d(e) = \sum_{k:e\in P_{T_k}(s,d)} \lambda_k, \qquad x_e = \sum_{k:e\in E(T_k)} \lambda_k.$$

Each f_d is an s–d flow of value $\sum_k \lambda_k$, while the load definition and feasibility of Λ give $f_d(e) \leq x_e \leq c_e$. The paths $P_{T_k}(s,d)$ also avoid $\delta^-(s)$ and $\delta^+(d)$. The resulting conceptual-flow solution therefore has the same multicast rate as Λ. ∎

The LP thus supplies a yardstick. With coding permitted at every relay, it would also supply the plan: the classical single-source multicast theorem makes ρ_{cf} attainable directly over its edge flows [17], and no tree needs to be constructed at all. However, as a pragmatic design choice that makes *Spring* much easier to be deployed, we keep relays replication-only. In this case, an executable plan must instead be an explicit family of receiver-covering trees, and searching for one reintroduces an NP-hard directed Steiner problem in the

pricing step below. Fountain coding using RaptorQ, to be soon introduced in Section IV-D, does not change this difficulty, nor does it substitute for coding at relays; it acts only after the trees are installed, making symbols interchangeable across them so that a temporarily slow tree delays only its own in-flight symbols.

B. Tree Realization

For any edge-budget vector b, let $\rho_{\text{pack}}(b)$ be the maximum rate carried by a fractional packing of receiver-covering trees. For each tree $T \in \mathcal{T}$, define $a_{eT} = 1$ when T contains edge e and $a_{eT} = 0$ otherwise. The packing LP is

$$\rho_{\text{pack}}(b) = \max_{\{\lambda_T\}_{T\in\mathcal{T}}} \quad \sum_{T\in\mathcal{T}} \lambda_T \tag{6}$$

$$\text{s.t.} \quad \sum_{T\in\mathcal{T}} a_{eT}\lambda_T \le b_e, \qquad \forall e \in E, \tag{7}$$

$$\lambda_T \ge 0, \qquad \forall T \in \mathcal{T}.$$

The source-side egress-relay structure yields a sufficient condition under which this packing reaches the conceptual-flow bound. For each $a \in \mathcal{A}$, recall that $e_a = (s, a)$ is its source-side egress-relay edge.

Proposition 2 (Source-side egress-relay sufficiency)**.** *Suppose the packing ranges over all receiver-covering trees without a hop limit, and every source-side egress-relay edge satisfies*

$$c_{e_a} \ge \sum_{r\in\mathcal{R}:(a,r)\in E} c_{(a,r)}, \qquad \forall a \in \mathcal{A}. \tag{8}$$

Then the full measured capacity vector admits a receiver-covering tree packing of rate ρ_{cf}; equivalently,

$$\rho_{\text{pack}}(c) = \rho_{\text{cf}}.$$

Proof: Let $U = \{s\} \cup \mathcal{A}$ denote the source cluster. Decompose each receiver-specific flow of an optimal conceptual-flow solution into s–d paths after cycle removal. For every path, retain the suffix that begins at its last vertex in U, then contract U into one root σ, preserving edges that leave U as labeled parallel edges. The retained paths preserve rate ρ_{cf} to every receiver, and no receiver-specific flow increases on an edge outside U.

After contraction, every non-root vertex is a receiver. Hence every nonempty $X \subseteq \mathcal{R}$ has incoming capacity at least ρ_{cf}: a σ–d flow of that value crosses the cut for any $d \in X$. The capacitated form of Edmonds' arborescence-packing theorem therefore gives a fractional packing of σ-rooted spanning arborescences with total rate at least ρ_{cf} [37], [38]; retain a subpacking of rate ρ_{cf}.

To lift an arborescence back to $\mathcal{G}$, restore the original tail s or a of every edge leaving σ and add $e_a = (s, a)$ whenever the tree uses an edge from relay a to a receiver. All other edge loads remain unchanged. The added load on e_a satisfies

$$\sum_{T:e_a\in E_T} \lambda_T \le \sum_{r\in\mathcal{R}:(a,r)\in E} \sum_{T:(a,r)\in E_T} \lambda_T \le \sum_{r\in\mathcal{R}:(a,r)\in E} c_{(a,r)} \le c_{e_a}.$$

The lifted family is therefore capacity feasible and has rate ρ_{cf}. Proposition 1 supplies the matching upper bound. ∎

Eq. (8) is a sufficient condition. It requires each trainer-to-relay connection to match the combined capacities of that relay's wide-area connections, even though the relay receives each symbol once and can copy it to several receivers. When no egress relays are present, the condition disappears and the proposition applies directly to the source-only and receiver-forwarding configurations. An overlay that does not satisfy the condition may still attain the conceptual-flow bound. The packing computation and per-instance check below determine whether the family returned by *Spring* reaches it.

Spring solves the packing LP with $b = c$, letting the measured edge capacities constrain the explicit trees directly, and compares the returned packing rate with the conceptual-flow target. The final packing LP returns only the trees with nonzero planned rates; these form the finite family Λ defined in Section III.

Spring verifies the returned family against both the target rate and the measured edge capacities:

$$\rho(\Lambda) = \sum_{k=1}^{K} \lambda_k = \rho_{\text{cf}}, \tag{9}$$

$$\sum_{k:e\in E(T_k)} \lambda_k \le c_e, \qquad \forall e \in E. \tag{10}$$

Eq. (9) checks the rate delivered to the full receiver set, and Eq. (10) checks the load placed on every overlay edge. The following proposition states that a family passing both checks explicitly realizes the conceptual-flow optimum in the fractional planning model.

Proposition 3 (Checked realization)**.** *If a returned family Λ satisfies Eqs. (9)–(10), then*

$$\rho_{\text{pack}}(c) = \rho_{\text{cf}}.$$

Proof: Every tree in Λ reaches every receiver, so Eq. (9) delivers rate ρ_{cf} to the full receiver set. Eq. (10) keeps its edge loads within c, giving $\rho_{\text{pack}}(c) \ge \rho_{\text{cf}}$. Proposition 1 gives the matching upper bound. ∎

The tree set $\mathcal{T}$ is exponential, so *Spring* solves the packing LP by column generation [35]. The restricted master problem assigns fractional rates to the trees found so far. Its dual assigns a nonnegative price π_e to each edge budget, and the pricing step searches for a receiver-covering tree of minimum total price, $\min_{T\in\mathcal{T}} \sum_{e\in E} a_{eT}\pi_e$. A tree with price below one improves the restricted master and is added as a new column. When exact pricing proves that every remaining tree has price at least one, the current master solution is optimal for the full packing LP. The pricing problem is a directed Steiner arborescence rooted at s with terminal set $\mathcal{R}$ [36], and is NP-hard in general [39]. *Spring* implements this exact pricing oracle with an integer program; the conceptual-flow and restricted-master formulations remain linear programs. When a deployment imposes a hop limit, both the pricing guarantee and the resulting family are relative to that restricted tree set.

This procedure reports ρ_{cf} only for a returned family that passes Eqs. (9)–(10). A smaller returned rate is reported as

a feasible lower bound, without a realization claim for the conceptual-flow optimum.

Section V-B evaluates this sequence on generated overlays that satisfy Eq. (8), then reports whether the families constructed by *Spring* pass Eqs. (9)–(10) under the measured capacities c.

Installing the tree family. To install the trees, the planner serializes each selected tree as an identifier and a directed edge list. The controller validates these descriptions and installs tree-specific forwarding entries at the participating nodes. Each entry maps a tree identifier to the downstream neighbors that should receive a copy of a symbol carried on that tree. Only these entries enter the data plane; the fractional rates λ_k remain planning variables used to account for capacity and certify $\rho(\Lambda)$, and the runtime does not enforce them as pacing weights.

C. Python Service Interface

Spring uses FFI bindings to expose its Rust implementation through a thin Python API for multicast-group management and exact-object delivery. Applications can create a multicast group, join or leave it, install a planner-supplied tree family, and initiate a lossless transfer from an in-memory object. Receivers can reconstruct the object in memory or write reconstructed block data directly to a file, avoiding an object-sized in-memory receive buffer for multi-GB artifacts. Each send or receive returns a session identifier through which the application waits for completion. Tree-specific forwarding, symbol scheduling, and feedback rounds remain inside *Spring*.

D. Lossless Multi-Tree Delivery

Spring combines a lossless data path with source-side RaptorQ coding and receiver feedback so that the installed trees can collaborate on one object. Crucially, session data symbols are non-droppable. When a bounded queue fills, backpressure preserves an admitted symbol until downstream capacity becomes available. *Spring* uses RaptorQ so that symbols delivered over different trees can help reconstruct the same part of the object. A faster tree can therefore keep delivering useful symbols while another symbol remains in flight on a slower tree. Receiver feedback identifies incomplete blocks, and *Spring* estimates how many additional coded symbols each block still needs. The sender generates extra coded symbols until every participating receiver has reconstructed the object. A timeout or abort reports failure and does not return an incomplete object. Fig. 2 summarizes this session and shows how a receiver combines symbols delivered over different trees.

Session establishment. The session begins with the lightweight MANIFEST–READY handshake shown in Fig. 2(a). In the MANIFEST, the sender announces the object size, its source-block layout, and the installed tree identifiers. Each receiver prepares to receive the object and replies READY. Once the participating receivers are ready, *Spring* begins data transmission and keeps the receiver set fixed throughout the session. Payload symbols carry tree identifiers, whereas the MANIFEST, READY, feedback, and round markers belong to the session as a whole.

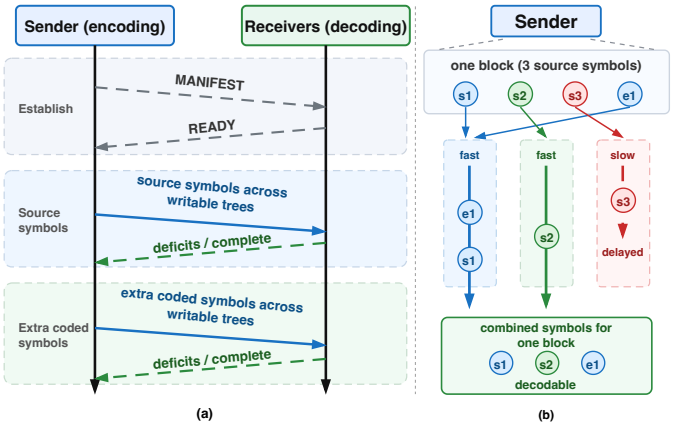


Fig. 2. *Spring*'s lossless multi-tree object delivery. (a) Session establishment and feedback-driven sending rounds. (b) At the receiver, symbols delivered over different trees form a decodable set for one RaptorQ source block.

RaptorQ blocks and symbols. During data transmission, *Spring* sends the source symbols of each RaptorQ source block in the initial round. In subsequent rounds, *Spring* generates extra coded symbols for a block whenever at least one receiver cannot yet reconstruct it. Because RaptorQ is systematic, a receiver that obtains all source symbols restores their original bytes directly. When extra coded symbols are needed, *Spring* constructs the block's RaptorQ encoder on first use and reuses it across rounds: the encoder precodes the block once into intermediate symbols [11], from which each extra coded symbol is generated on demand, without re-encoding the block's source data.

In *Spring*, each transmitted symbol carries identifiers for its source block and the tree selected for its delivery. A forwarding node uses the tree identifier to select the downstream children and copies the symbol to them unchanged. A receiver instead combines all distinct symbols belonging to the same block, regardless of which tree delivered them. For source block b, let $H_{b,k,d}(t)$ be the set of distinct symbols that receiver d has received through tree T_k by time t. The receiver combines the symbols delivered over all installed trees, $H_{b,d}(t) = \bigcup_{k=1}^{K} H_{b,k,d}(t)$, and reconstructs the block when the RaptorQ backend declares $H_{b,d}(t)$ decodable. A symbol's tree identifier determines how it is delivered, while every useful symbol contributes to the same block-level decoding process.

Combining symbols in this way avoids assigning a fixed portion of a block to each tree. Faster trees can therefore continue advancing block reconstruction when another tree slows. Once the decoder reconstructs the block, the receiver writes its recovered bytes back to the block's position in the object. RaptorQ encoding occurs only at the source. An inference worker can play both roles: it contributes a received symbol to its own block decoder and copies the unchanged symbol to downstream receivers.

Backpressure-driven sending. During each sending round, *Spring*'s unweighted, backpressure-aware round-robin sched-

uler offers the next symbol to the next installed tree. If that tree cannot currently accept the symbol, the scheduler tries the next tree; a symbol becomes assigned only when a tree accepts it. Once assigned, the symbol is replicated along that tree to every receiver. When a downstream queue fills, backpressure propagates toward the sender along the tree. The assigned symbol remains buffered rather than being dropped, while the tree becomes temporarily unwritable for subsequent symbols and the scheduler continues through another writable tree. The same procedure carries source symbols in the initial round and extra coded symbols in subsequent rounds. It keeps the sender working whenever at least one tree is writable, without changing the installed forwarding paths.

Feedback and completion. As shown in Fig. 2(a), receiver feedback determines which extra coded symbols *Spring* sends after the initial round. At the end of each sending round, every receiver reports either that it has reconstructed the complete object or, for each incomplete block, a symbol deficit estimating how many additional coded symbols it still needs for reconstruction. Since every extra coded symbol is delivered to all receivers, the sender tracks the largest deficit reported for each block and generates that many extra coded symbols for it in the current round. The sender assigns these symbols to currently writable trees. After receiving the extra coded symbols, receivers report their updated status. If any receiver still cannot reconstruct a block, *Spring* begins another feedback round. The session completes when every receiver reports completion.

Fig. 2(b) illustrates a block with three source symbols, $\{s_1, s_2, s_3\}$, delivered over two fast trees and one slow tree. Suppose each tree buffers one pending symbol. The round-robin scheduler assigns s_1 and s_2 to the fast trees and s_3 to the slow tree. The queued s_3 triggers backpressure and makes the slow tree temporarily unwritable. At the end of the source-symbol round, the receiver has s_1 and s_2, with s_3 still in flight, and reports a one-symbol deficit. The sender generates an extra coded symbol e_1, which the scheduler assigns to a writable fast tree. The receiver obtains e_1 before s_3, and $\{s_1, s_2, e_1\}$ forms a decodable set. It reconstructs the block without waiting for s_3; the slow tree delays only its in-flight symbol and owns no fixed portion of the block.

V. Evaluation

Our evaluation follows *Spring* from multicast planning through WAN object delivery. We first test whether the planner returns explicit tree families that pass the realization check. We then study tree pruning, user-space buffering, source-side egress relay scaling, and adaptation to bandwidth changes. Finally, we evaluate the complete system on a seven-receiver WAN, including the transfer of an RL policy artifact.

A. Methodology and Metrics

The planner study runs only the control-plane optimization on generated directed overlays, using a MacBook Pro with an Apple M3 Pro (11-core CPU) and 18 GB of memory. *Spring* supports both deployment modes used in the runtime experiments. In the single-machine experiments, the controller runs locally and each dataplane node runs in an isolated Linux network namespace. Hierarchical token bucket (HTB) traffic control enforces the overlay-edge capacities. In the WAN experiments, the controller and dataplane processes run in Docker containers on geographically distributed physical machines. Before each WAN planning run, we concurrently probe the candidate overlay connections. For every edge $e = (u, v)$, node v measures the transfer rate of a fixed-volume TCP probe from u and reports it as c_e. We assemble these rates into a directed capacity matrix and compute the corresponding tree family. Since the probe follows the current underlay path, c_e reflects its routing, shaping, and competing traffic.

Unless stated otherwise, single-machine transfers use seven receivers, a 500 MiB synthetic object, reliable TCP, 128 KiB RaptorQ source blocks, and 16 source symbols per block.

We compute runtime throughput from the delivery interval defined in Section III. For receiver d, τ_d spans from its first accepted payload symbol until every block is reconstructed, and $\tau = \max_d \tau_d$. The *extra-symbol ratio* is the number of symbols accepted for transmission beyond the source-symbol count, divided by that count. It upper-bounds wire overhead because a session may finish before every accepted symbol is transmitted.

Unless stated otherwise, results average three single-machine runs or five WAN runs, and whiskers show one standard deviation. Every included run completes at all receivers and passes payload-hash verification. Extra-symbol ratios use one detailed-log run.

B. Planning and Tree Realization

We begin with the planner because the conceptual-flow target becomes executable only through an explicit tree family. To isolate this control-plane step, we generate role-constrained directed overlays with four forwarding configurations: *source only*, *source-side egress relays*, *receiver forwarding*, and *both*. Every configuration contains the same direct edges from the source to the receivers. The egress-relay configurations add four nodes in $\mathcal{A}$, together with their source-side egress-relay edges and outgoing edges to every receiver. The receiver-forwarding configurations add directed edges among the inference workers.

We draw the capacities of all wide-area edges from a clipped lognormal distribution with median 200 Mbps and range 20–1000 Mbps. For each receiver count and random seed, a logical edge retains the same capacity in every configuration that contains it, so the comparison adds forwarding roles to a paired base overlay. We set each source-side egress-relay edge to satisfy Eq. (8) with equality. We vary the receiver count from four to twelve in steps of two and use ten seeds, giving 200 planner instances across the four configurations.

By construction, all 200 instances satisfy Eq. (8). Proposition 2 therefore guarantees that a receiver-covering tree packing of rate ρ_{cf} exists under c. We run *Spring*'s planner under c and apply the realization check. The study records ρ_{cf}, solve time, certification status, and the number of trees

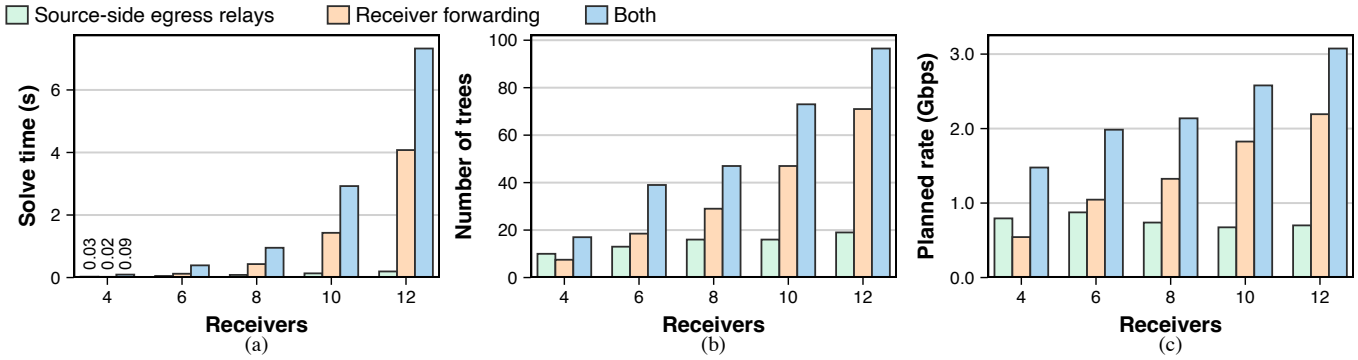


Fig. 3. Planner results on role-constrained directed overlays. Four source-side egress relays are used alone or with receiver forwarding; source-only yields one star tree and is omitted. Bars show medians over ten paired capacity assignments: (a) solve time, (b) the number of trees returned, and (c) planned multicast rate.

returned by the final packing LP. Each returned tree has a nonzero planned rate.

Fig. 3 shows the structural tradeoff. Adding source-side egress relays, receiver forwarding, or both returns more trees and raises the planned rate over source-only fanout. At 12 receivers, receiver forwarding and the combined configuration reach median planned rates of 2.19 and 3.07 Gbps, respectively, compared with 52 Mbps for source-only fanout. The additional forwarding structure also increases planning cost: the combined configuration reaches a median solve time of 7.32 s at 12 receivers, while all other reported medians are lower.

All 200 returned families pass *Spring*'s realization check under c, and all 200 runs terminate with an exact-pricing optimality certificate. The maximum relative difference between $\rho(\Lambda)$ and ρ_{cf} stays below 10^{-14}, within numerical precision. The proposition supplies the existence result; the returned trees and their verified edge loads provide a constructive witness for each tested instance. For the measured WAN and other overlays, *Spring* applies the same check to each full returned family.

The check establishes a capacity-feasible fractional tree packing. Whether every returned tree helps finite-object execution remains a runtime question. We next study how tree pruning and user-space buffer capacity shape runtime throughput.

C. Runtime Configuration

At runtime, *Spring* visits writable trees in round-robin order; the planned rates λ_k are not pacing weights, so a tree with a small planned contribution receives scheduling opportunities like any other. If it blocks, its assigned symbols remain queued while other trees continue delivering, and feedback may trigger extra coded symbols for the affected blocks. Pruning reduces this runtime overhead while also removing some capacity and path diversity.

The second choice is the user-space buffer capacity C, measured in symbols. These buffers hold symbols awaiting outgoing transmission. Once admitted, a symbol remains assigned to its tree. A full buffer propagates backpressure upstream, allowing subsequent symbols at the sender to use other writable trees. We apply the same C at every emulated node and vary it together with the pruning threshold.

The experiment uses one fixed seven-receiver overlay with three source-side egress relays and receiver forwarding enabled. The certified family used in this experiment contains 44 trees with nonzero planned rates. We install all 44 trees, or remove trees whose planned rate is below 1, 5, 10, 20, or 30 Mbps before installation. For each of these six pruning thresholds we sweep $C \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$, yielding 54 configurations and 162 verified transfers. The payload and transport otherwise follow Section V-A.

Fig. 4(a) shows that selective pruning improves finite-transfer throughput. At $C = 1$, removing trees below 5 Mbps reduces the installed set from 44 to 37 while preserving 98.3% of its planned rate; throughput rises from 741 ± 4 to 763 ± 3 Mbps. A 10 Mbps threshold performs similarly, whereas 20 and 30 Mbps remove enough path diversity to lower throughput. The 5–10 Mbps range therefore balances runtime work against the capacity retained in the installed family.

Panels (b) and (c) show why shallow buffers complement that pruning. As C grows, more symbols remain committed to a tree before backpressure redirects new work. For the 5 Mbps tree set, increasing C from 1 to 256 raises the extra-symbol ratio from 3.1% to 110.4% and lowers throughput from 763 to 594 Mbps. All six thresholds show the same association between deeper buffers, more extra symbols, and lower throughput.

These user-space buffers sit above TCP, which independently transports bytes over each connection. The best observed configuration uses the 5 Mbps threshold and $C = 1$; we retain it for the relay-scaling study.

D. Scaling Source-Side Egress Relays

Source-side egress relays expose additional WAN egress at the source datacenter, while the source still feeds each relay over its source-side egress-relay edge. We vary the relay count to identify when this additional fanout ceases to improve throughput. The experiment uses a nested family of seven-receiver overlays: existing edges retain their capacities, and each added relay contributes only its own new edges. For

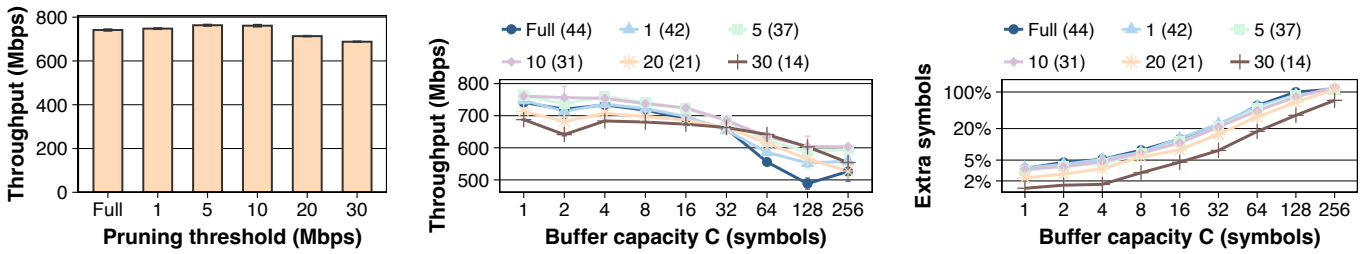


Fig. 4. Single-machine runtime configuration. (a) Throughput at $C = 1$ versus pruning threshold. For the same thresholds, (b) throughput and (c) extra-symbol ratio versus buffer capacity C.

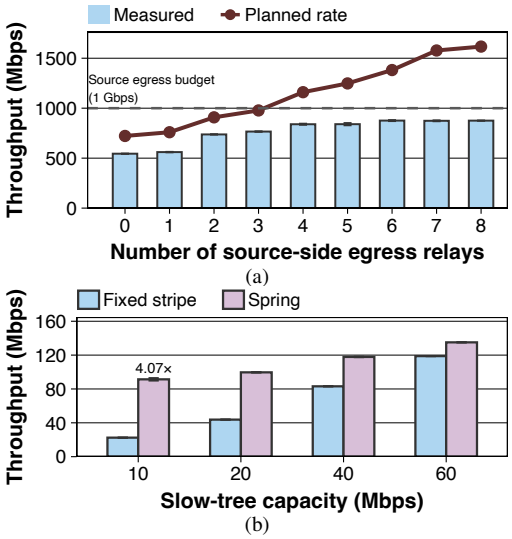


Fig. 5. Single-machine experiments. (a) Throughput as source-side egress relays are added. (b) Throughput after one of two initially 80 Mbps trees slows five seconds into the transfer.

$|\mathcal{A}| \in \{0, \ldots, 8\}$, we compute a new tree family, prune trees below 5 Mbps, and run with $C = 1$.

The planner constrains overlay edges independently, while the single-machine execution also enforces a 1 Gbps aggregate ingress and egress budget at every emulated node. Comparing the planned rate with measured throughput reveals when the source's aggregate egress, which lies outside the edge-based planning model, becomes limiting.

Fig. 5(a) reveals both regimes. With no source-side egress relay, *Spring* reaches 545 Mbps. Throughput rises to 839 Mbps with four relays and 876 Mbps with six, then remains between 874 and 876 Mbps. Meanwhile, the planned rate continues to grow, reaching 1.62 Gbps with eight relays. Once it exceeds the source's 1 Gbps aggregate egress budget, additional edge-level capacity no longer raises measured throughput.

E. Adapting to Bandwidth Changes

The preceding relay sweep holds bandwidth fixed within each transfer. We now examine how *Spring* responds when the bandwidth available to one installed tree changes during delivery. Fixed striping assigns each portion of an object to one tree, so a mid-transfer slowdown delays that portion even when another tree has available capacity. *Spring* lets every writable tree continue contributing symbols to the same RaptorQ object.

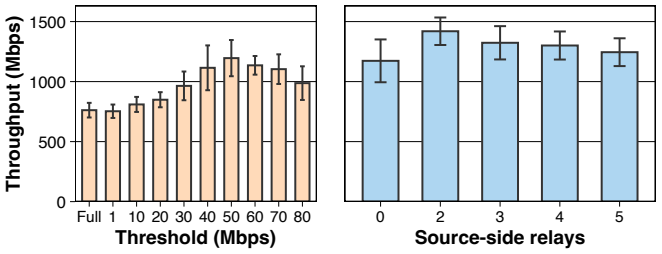
To emulate a mid-transfer bandwidth change while running *Spring*'s actual dataplane, we use two receiver-covering trees and two receivers. Both trees begin at 80 Mbps. Five seconds after payload transfer begins, an in-place HTB class change reduces one tree to 60, 40, 20, or 10 Mbps. The sender keeps running, the tree family remains fixed, and the TCP connections are neither closed nor rebuilt. Because the initial rates are equal, the fixed-stripe baseline assigns eight of 16 stripes to each tree before transmission and never revisits that assignment. *Spring* uses the same two trees to deliver a 500 MiB RaptorQ object with round-robin scheduling, per-tree backpressure, and $C = 1$.

Even the mild 80-to-60 Mbps change gives *Spring* a 14% throughput advantage, as Fig. 5(b) shows. The advantage grows with the severity of the slowdown: *Spring* is $1.42\times$ faster at 40 Mbps, $2.28\times$ at 20 Mbps, and $4.07\times$ at 10 Mbps. At the last point, fixed striping reaches only 22.5 Mbps because half of the object remains tied to the slow tree; *Spring* reaches 91.4 Mbps because the 80 Mbps tree continues to carry symbols that advance reconstruction of that object. Its extra-symbol ratio remains below 0.7% throughout this sweep, so the gain does not require a large increase in extra coded symbols.

Both approaches use the same two-tree topology over reliable TCP and experience transport backpressure; no packet loss is injected, and every Linux traffic-control link reports zero drops throughout the experiment. The gain therefore comes from directing subsequent useful symbols to writable trees as bandwidth changes, while the installed tree family stays fixed. Before a subsequent transfer, *Spring* can re-probe the overlay and compute a new tree family from the updated measurements.

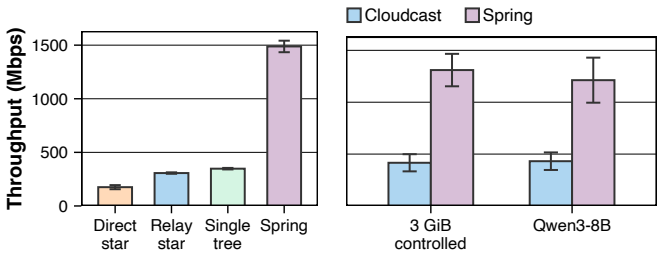
This division of labor also marks the prototype's measurement boundary: one probe round, one capacity matrix, and one solve per session, with the installed family fixed for the transfer. The prototype attaches no lifetime to its measurements, detects no drift, and supports no acknowledged mid-transfer route update; at larger scales, a production deployment would need recurring probes, aged-out estimates, and re-planned families installed between policy updates.

To determine whether these mechanisms translate into WAN gains on a geographically distributed deployment, we evaluate *Spring* over a real WAN.



(a) (b)

Fig. 6. WAN configuration. (a) Tree-pruning threshold with two egress relays; "Full" installs all trees. (b) Source-side egress relay count at the selected 50 Mbps threshold.



(a) (b)

Fig. 7. WAN throughput. (a) Fixed topologies and Spring for 3 GiB. (b) Cloudcast and Spring for 3 GiB and a 16.39 GB Qwen3-8B artifact.

F. WAN and Policy-Object Delivery

Our WAN deployment uses one trainer and up to five source-side egress relays on six machines of the Arbutus research cloud in Victoria, together with seven DigitalOcean dedicated-CPU c-4 receivers in AMS, LON, TOR×2, SFO2, and SFO3×2. Receivers may forward for one another, and each process runs in a Docker container with host networking. Each comparison below is one measurement campaign with the same placements and object; WAN conditions shift between campaigns, so absolute throughput varies across figures while within-figure comparisons remain matched. We first tune the tree-pruning threshold and source-side egress-relay count for this deployment.

Fig. 6(a) shows that pruning must be retuned for the WAN: its measured capacities shift the cost–capacity balance well above the single-machine setting. Installing all 31–33 trees in the certified families reaches 762 ± 61 Mbps. Raising the threshold improves throughput until 50 Mbps, where 8–15 trees reach 1196 ± 151 Mbps. At 80 Mbps, only 3–7 trees remain and throughput falls to 987 ± 140 Mbps. We therefore remove trees with planned rates below 50 Mbps in the remaining comparisons and report the measured delivery throughput of the resulting family. The realization certificate applies to the full, unpruned family.

Fig. 6(b) revisits relay scaling on the physical deployment. Two source-side egress relays raise throughput from 1173 ± 178 Mbps without relays to 1420 ± 115 Mbps: the per-receiver copies are now paid from the relays' pooled wide-area egress rather than the trainer's own. With five relays, however, measured throughput falls to 1245 ± 116 Mbps even as the unpruned planned rate keeps rising: relays cannot raise any receiver's ingress, and the larger families add runtime work, echoing Section V-D and Section V-C. We therefore use two relays in the remaining comparisons.

Fig. 7(a) asks whether a planned tree family matters once the runtime and deployment are fixed. Direct fanout sends one copy from the trainer to each receiver, the relay star uses a fixed two-relay fanout, and the single-tree baseline uses the highest-rate measured receiver-covering tree. These baselines reach 175–348 Mbps. From the same measured overlay, *Spring* installs 14 trees and reaches 1488 ± 54 Mbps, $4.3\times$ the strongest fixed-topology baseline. This comparison captures the combined effect of the planned tree family and coded delivery across that family.

Fig. 7(b) next compares *Spring* with Cloudcast, which binds fixed stripes to planned trees [10]. Our Cloudcast arm plans from the same probe round and runs on the same data plane: 16 stripes bind to 10–13 receiver-covering trees under Cloudcast's hop constraint, with no coding and no cross-tree repair. Cloudcast reaches 416 ± 83 Mbps, while *Spring* reaches 1310 ± 156 Mbps by letting every installed tree contribute to one RaptorQ object, a $3.1\times$ gain.

We finally evaluate policy-object delivery with a 16.39 GB Qwen3-8B [3] artifact produced by a group relative policy optimization workload [2]. Across five runs, *Spring*'s mean delivery time is 110.9 s and its mean throughput is 1212 ± 217 Mbps. The corresponding Cloudcast means are 313.3 s and 431 ± 85 Mbps. This experiment measures the delivery of one exact policy version; model quality lies outside its scope.

VI. Concluding Remarks

For two decades, the ingredients of optimal overlay multicast have existed as point solutions: conceptual flows bound what an overlay can deliver, tree packing turns capacity into forwarding structure, and fountain codes free delivery from any particular path. *Spring* is, to our knowledge, the first open-source system that integrates all of them into one working service. Its planner treats the conceptual-flow optimum not as an abstraction but as a planning target to be met: it constructs an explicit family of receiver-covering trees and checks, instance by instance, that the family attains that optimum, with relays that replicate only. Its runtime then repurposes fountain coding from erasure correction to cross-tree collaboration: faster and slower trees jointly deliver the same object, a backlogged tree delays only its own in-flight symbols, and every symbol carried by a writable tree advances every receiver's decoding.

The design that emerges is strikingly simple: there are no per-tree rates to enforce, no bandwidth to predict, and no fixed binding of data to trees. Instead, an unweighted round-robin and per-tree backpressure suffice. On a real wide-area deployment, *Spring* outpaced the best fixed topology by $4.3\times$. Geo-distributed RL post-training has made the classic problem of maximizing multicast throughput urgent again, and *Spring* answers it with theory, protocol design, and practical implementation in equal measure. Two decades of theory told us how fast overlay multicast could be; *Spring* — in Rust, behind a simple Python API, and open source — is our attempt to actually go that fast, and an invitation for others to go faster.

REFERENCES

[1] B. Wu, S. Wang, Y. Tang, J. Ding, E. Helenowski, L. Tan, T. Xu, T. Gowda, Z. Chen, C. Zhu, X. Tang, Y. Qian, B. Zhu, and R. Hou, "LlamaRL: A Distributed Asynchronous Reinforcement Learning Framework for Efficient Large-scale LLM Training," *arXiv preprint arXiv:2505.24034*, 2025.

[2] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo, "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models," *arXiv preprint arXiv:2402.03300*, 2024.

[3] Qwen Team, "Qwen3 Technical Report," *arXiv preprint arXiv:2505.09388*, 2025.

[4] Prime Intellect Team, S. Jaghouar, J. Mattern, J. M. Ong, J. Straube, M. Basra, A. Pazdera, K. Thaman, M. Di Ferrante, F. Gabriel, F. Obeid, K. Erdem, M. Keiblinger, and J. Hagemann, "INTELLECT-2: A Reasoning Model Trained Through Globally Decentralized Reinforcement Learning," *arXiv preprint arXiv:2505.07291*, 2025.

[5] The PyTorch Team at Meta, "Introducing torchforge – A PyTorch Native Library for Scalable RL Post-Training and Agentic Development," 2025. [Online]. Available: https://pytorch.org/blog/introducing-torchforge/

[6] A. Riabov, "Efficient Information Dissemination Systems," Ph.D. dissertation, Columbia University, 2004. [Online]. Available: https://www.columbia.edu/~dano/theses/riabov.pdf

[7] Y. Cui, Y. Xue, and K. Nahrstedt, "Max-min Overlay Multicast: Rate Allocation and Tree Construction," in *Proc. 12th IEEE International Workshop on Quality of Service (IWQoS)*, 2004, pp. 221–231.

[8] M. Wang, Z. Li, and B. Li, "A High-Throughput Overlay Multicast Infrastructure with Network Coding," in *Proc. 13th International Workshop on Quality of Service (IWQoS)*, 2005, pp. 37–53.

[9] Z. Li, B. Li, D. Jiang, and L. C. Lau, "On Achieving Optimal Throughput with Network Coding," in *Proc. 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2005, pp. 2184–2194.

[10] S. Wooders, S. Liu, P. Jain, X. Mo, J. E. Gonzalez, V. Liu, and I. Stoica, "Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud," in *Proc. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024, pp. 281–296.

[11] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RFC 6330: RaptorQ Forward Error Correction Scheme for Object Delivery," 2011.

[12] X. Zhang, S. Chang, and B. Li, "Nextmini: A New Research Testbed for Network Emulation and Experimentation," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2026, pp. 1–10.

[13] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," in *Proc. 35th International Conference on Machine Learning (ICML)*, 2018, pp. 1407–1416.

[14] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed Prioritized Experience Replay," in *Proc. 6th International Conference on Learning Representations (ICLR)*, 2018.

[15] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for Distributed Reinforcement Learning," in *Proc. 35th International Conference on Machine Learning (ICML)*, 2018, pp. 3053–3062.

[16] Z. Li, B. Li, and L. C. Lau, "On Achieving Maximum Multicast Throughput in Undirected Networks," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2467–2485, 2006.

[17] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[18] S.-Y. R. Li, R. W. Yeung, and N. Cai, "Linear Network Coding," *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[19] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *Proc. ACM SIGCOMM*, 1998, pp. 56–67.

[20] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[21] M. Luby, "LT Codes," in *Proc. 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002, pp. 271–280.

[22] A. Shokrollahi, "Raptor Codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[23] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 767–780, 2004.

[24] Y.-h. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A Case for End System Multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1456–1471, 2002.

[25] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr., "Overcast: Reliable Multicasting with an Overlay Network," in *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000, pp. 197–212.

[26] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination," in *Proc. 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2001, pp. 11–20.

[27] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments," in *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 298–313.

[28] D. Kostić, A. Rodriguez, J. R. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh," in *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 282–297.

[29] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.

[30] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[31] M. Noormohammadpour, C. S. Raghavendra, S. Rao, and S. Kandula, "DCCast: Efficient Point to Multipoint Transfers Across Datacenters," in *Proc. 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2017.

[32] M. Noormohammadpour, C. S. Raghavendra, S. Kandula, and S. Rao, "QuickCast: Fast and Efficient Inter-Datacenter Transfers Using Forwarding Tree Cohorts," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 225–233.

[33] L. Luo, L. Yu, T. Ma, and H. Yu, "Flexible and Efficient Multicast Transfers in Inter-Datacenter Networks," in *Proc. IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, 2022, pp. 1–10.

[34] S.-H. Tseng, S. Agarwal, R. Agarwal, H. Ballani, and A. Tang, "CodedBulk: Inter-Datacenter Bulk Transfers Using Network Coding," in *Proc. 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 15–28.

[35] P. C. Gilmore and R. E. Gomory, "A Linear Programming Approach to the Cutting-Stock Problem," *Operations Research*, vol. 9, no. 6, pp. 849–859, 1961.

[36] M. Charikar, C. Chekuri, T.-Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li, "Approximation Algorithms for Directed Steiner Problems," *Journal of Algorithms*, vol. 33, no. 1, pp. 73–91, 1999.

[37] J. Edmonds, "Edge-Disjoint Branchings," in *Combinatorial Algorithms*, ser. Courant Computer Science Symposium 9, 1973, pp. 91–96.

[38] H. N. Gabow and K. S. Manu, "Packing Algorithms for Arborescences (and Spanning Trees) in Capacitated Graphs," *Mathematical Programming*, vol. 82, no. 1, pp. 83–109, 1998.

[39] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, 1972, pp. 85–103.