Syntra: Synthesizing Cross-Layer Controllers for Low-Latency Video Streaming

Jia Pan, Anup Agarwal[#], Işıl Dillig, Venkat Arun
UT Austin, [#]CMU

Abstract

Modern applications such as low-latency video streaming demand tight coordination across multiple control dimensions, including bitrate selection, congestion control, frame skipping, and forward error correction (FEC). These dimensions interact in complex ways, making existing heuristic approaches difficult to design, tune, and generalize. This paper presents Syntra, an automated tool that synthesizes joint controllers from a symbolic model of the system and a declarative performance objective. Syntra formulates control as a partially observable game, performs bounded-horizon minimax search (similar to Chess engines) to synthesize strategies, and distills them into an efficient, interpretable policy via imitation learning. Synthesized controllers incorporate novel strategies that exploit the synergy between control dimensions to consistently outperform existing designs in our evaluation.

1 Introduction

Emerging applications like cloud gaming, AR/VR, and mobile robotics demand video streaming systems that deliver high quality with ultra-low latency. These workloads are unforgiving: a single missed frame deadline—caused by congestion, jitter, or slow encoding—can visibly degrade the experience, and there is often no time to retransmit or recover.

Meeting such tight latency budgets (e.g., 50-200 ms for cloud gaming [25, 51]) requires coordination across multiple layers of the stack, from adaptive bitrate (ABR) selection and frame skipping to congestion control and forward error correction (FEC). A high-bitrate frame may improve video quality, but only if it arrives on time; otherwise, it wastes capacity and increases latency. A more aggressive FEC configuration may reduce losses, but risks pushing future frames past their deadline.

Designing a good policy that jointly manages these tradeoffs is extremely difficult. The search space is large, the dynamics are uncertain, and key system variables—like link rate or queue occupancy—are hidden from the controller. Even after decades of work, there is still no consensus on how to design adaptive bit-rate (ABR) or congestion control algorithms (CCAs) even in isolation, let alone together. Prior work on cross-layer control (e.g., Salsify [20], VOXEL [38], AFR [35]) has shown that better coordination is possible, but such systems are hard to design by hand and fragile to changing requirements. As a result, prior work does not go beyond joint control of two layers. In many systems deployed today, the layers are only loosely coupled, if at all.

This paper presents Syntra, an automated tool that synthesizes robust, low-latency streaming controllers. Instead of hand-designing heuristics or training neural networks, Syntra takes a different approach: it synthesizes a complete control policy from a symbolic model of the system and a declarative performance objective. The user provides a high-level description of the network and encoder—e.g., constraints on link variability and encoder behavior—and Syntra generates a controller that satisfies the objective under all behaviors consistent with the model. Crucially, the model does not assume any probability distribution. As a result, Syntra produces controllers that are robust even against adversarial environments and remains effective under a wide range of real-world network conditions.

To achieve this goal, Syntra models control as a partially observable game, reasons symbolically about uncertainty, and uses planning to discover optimal strategies over a finite planning horizon. It then distills these strategies into compact, human-readable decision trees (e.g. Appendix A) that enable fast, interpretable execution at runtime. Unlike black-box optimization or learning-based approaches, it does not require trace data, faithful simulators or distributional assumptions.

Syntra systematically explores tens of millions of alternatives to uncover joint adaptation strategies—including novel forms of ABR, CCA, frame skipping, and FEC tuning—that are difficult to design by hand. Because Syntra enables full-stack co-design, the resulting joint controller shifts the Pareto frontier of achievable tradeoffs between quality, delay, and robustness, as we show in our evaluation.

We evaluate Syntra across a range of network conditions and application scenarios. Compared to strong baselines such as WebRTC-GCC [11], WebRTC-Vegas [8], and Salsify [20], Syntra improves P95 one-way delay by over 6×, increases median video quality (SSIM) by more than 2 dB, and maintains a higher of frame rate even under challenging conditions such as network paths with cellular networks, heavy ACK aggregation and short buffers. Notably, it outperforms baselines even on a simple link with a fixed bandwidth despite relying only on conservative, worst-case assumptions. The

synthesized controllers remain simple and interpretable (Pseudocode for synthesized controller is in **??**), enabling efficient real-time deployment without sacrificing performance.

Although this paper uses video streaming as a case study to illustrate the benefits of automated symbolic synthesis, we believe the underlying techniques are more broadly applicable. Machine-assisted design can achieve levels of performance and reliability that are difficult to attain through manual design alone.

Limitation. The generated controllers do not guarantee fairness between flows. This is because achieving provable starvation-freedom is hard; All existing delay-bounding CCAs experiencing extreme unfairness under paths that are common on the internet [4]. Further, flow-isolation—e.g., via rate limiting or fair queuing—is common on the internet, and long running flows rarely compete on the same bottleneck queue [9, 14, 22]. Nevertheless, fairness remains an important limitation for future work to address.

2 Related Work

Prior work on low-latency video streaming has explored a wide range of adaptation mechanisms, spanning congestion control, bitrate selection, reliability techniques, and frame skipping. Some systems optimize individual layers, while others attempt limited forms of joint optimization. In parallel, the literature on reactive synthesis explores principled methods for controller generation under uncertainty. We briefly review each of these areas and highlight how Syntra differs.

Joint controllers low-latency video. Where most past work focuses on one layer of a video-streaming system, the multi-dimensional nature of low-latency streaming has long been recognized [28], and several systems explore limited forms of joint optimization—combining CCA with ABR [20], ABR with frame skipping [35], CCA with FEC [31], ABR with FEC [13] and CCAs whose fairness properties adapt to streaming needs [36]. Despite this progress, prior work typically coordinates only two layers at a time, and design choices are often heuristic. In contrast, Syntra synthesizes joint control across all major dimensions—bitrate, congestion control, FEC, and frame skipping—using a principled, symbolic approach. We now survey the literature on layer-separated control algorithms.

Congestion control algorithms. Traditional CCAs fall into delay-based [6, 8, 10, 44], loss-based [19, 24, 41], and hybrid schemes [23, 43]. Algorithms specific to low-latency video streaming have also been developed, most famously Google's GCC [11].

ABR algorithms. Approaches for stored-video streaming [27, 32, 42, 50], including learning-based ones [33, 47], optimize bitrate selection based on available bandwidth, buffer occupancy, and known chunk sizes. Live video, however, introduces additional uncertainty because frames are generated on-the-fly and encoders do not respond well to rapid changes in the requested bit rate. Thus alternate approaches have been proposed [13, 20, 26].

Learning-driven controller synthesis. Learning-based methods have been widely used to synthesize controllers for various systems, including CCAs [2, 30, 45, 48, 49], ABR algorithms [47, 48], and FEC controllers [12]. These methods typically assume that the deployment environment exhibits behavior similar to the training distribution. Due to their sample inefficiency, training is often conducted in offline simulators, which fail to accurately reflect real-world conditions. In contrast, Syntra employs non-deterministic, non-stochastic models that make minimal assumptions about the environment (see §6.1); the resulting controllers remain robust under any environmental behavior—even an adversarial one—as long as it satisfies these assumptions. Online learning and planning methods have also been explored in transport and video domains [17, 18, 32, 34, 46]. Some of these approaches rely on strong environmental assumptions, similar to their offline counterparts, or adapt slowly due to reliance on online experimentation.

Program synthesis and reactive control. Our technical approach is related to *reactive synthesis* [39, 40], where the goal is to automatically construct controllers that satisfy temporal logic specifications under all possible input sequences. Classical reactive synthesis focuses on finite-state systems with full observability [40], while more recent work extends to infinite-state games with partial information. For example, Dimitrova et al. [16] develop synthesis techniques for two-player games with incomplete information and infinite symbolic states. Like these approaches, Syntra models the environment using a symbolic transition system in linear rational arithmetic (LRA) and seeks to ensure safety and reachability goals. However, rather than performing abstraction-refinement or unbounded symbolic synthesis, Syntra performs bounded-horizon planning over symbolic belief summaries and distills policies into compact decision trees. This enables scalable synthesis for high-dimensional real-time systems such as video streaming.

CCmatic. Our work is most closely related to CCmatic [3] which synthesizes CCAs from a symbolic model. Syntra improves upon CCmatic's belief bound calculation (see § 5.2) and develops a faster synthesis strategy. Where CCmatic takes a week to synthesize a CCA that controls only the sending rate, Syntra can synthesize that CCA in minutes and a joint controller over five decisions in 3-4 hours.

3 Lessons Learned about Video Streaming using Syntra

This section illustrates how Syntra synthesizes performant controllers for real-time video streaming. Rather than describing the internals of the synthesis framework (covered in Sections 4–5), we focus here on what Syntra enables: the

discovery of non-obvious cross-layer strategies that emerge directly from declarative specifications. Through examining the controllers synthesized by Syntra, we identify a set of subtle interactions across control decisions and objectives that would be difficult to uncover through manual design. We also highlight conjectured *fundamental limits* revealed during this exploration, providing insights into how video streaming systems must balance latency, quality, and uncertainty.

3.1 Video Streaming Setting

We view the video streaming system as being structured into a joint control plane and a modular data plane. The data plane uses standard components: a video encoder, Reed-Solomon FEC encoder, and a UDP-based transport stack. It reports three quantities to the controller every round trip time (RTT): the number of bytes acknowledged and lost, and the size of the encoded frames. In return, the controller makes five decisions for the data plane:

- Whether to skip encoding the next frame,
- The target bitrate for encoding,
- The number of FEC packets to add,
- How to group frames for FEC,
- The send rate over the network.

Designing such a controller is challenging because its decisions interact and must jointly balance conflicting objectives. For instance, achieving long-term performance requires sending substantial traffic to infer the bottleneck link rate and buffer size—quantities that are not directly observable. However, this can compromise short-term goals such as minimizing latency and avoiding stalls.

Syntra addresses this challenge by efficiently exploring the combinatorial space of actions and their consequences to uncover novel strategies that exploit subtle interactions and tradeoffs. These are too complex for human designers, which is the key reason why it outperforms baselines—even on simple network scenarios.

3.2 Interactions between the Layers

Designing an effective video streaming controller requires navigating a complex landscape of cross-layer interactions. Below, we summarize several key tradeoffs that emerged from analyzing Syntra-generated controllers. Each highlights a non-trivial interaction among control decisions, illustrating why manual design is difficult and why automatic synthesis is necessary to exploit them. Section § 7.3 will illustrate them with concrete traces, using the ❶ notation to reference this section.

❶ **FEC grouping ⇔ latency vs. throughput tradeoff.** Reed–Solomon erasure codes encode k data packets into $n > k$ packets, enabling recovery from up to $t = n - k$ losses. The way video frames are grouped for FEC introduces a tradeoff between latency and throughput. If each frame is encoded into its own block, it can be decoded as soon as enough packets for that frame arrive. However, this is inefficient: to tolerate t losses, each frame must be padded with t redundant packets. Grouping multiple frames reduces redundancy overhead but increases FEC decoding delay. Syntra revealed that one can dynamically adjust FEC grouping based on network variability to achieve both low latency and high throughput—a behavior that static grouping policies miss.

❷ **FEC amount ⇔ bandwidth probe.** To estimate network capacity, the controller must occasionally send at rates above its current rate. However, doing so risks packet loss, which is particularly damaging in video streaming. If a frame is lost, the decoder cannot decode subsequent frames that depend on it, delaying playback. To avoid this, the controller must add enough FEC to guarantee recovery, and this amount needs to be calculated by the controller based on recent network conditions. Overall, Syntra discovered that successful probing requires jointly planning send rate and FEC redundancy to ensure reliable exploration without causing frame loss.

❸ **FEC amount ⇔ latency bound.** Adding excessive FEC increases the total number of packets to send, which can introduce queuing delays and cause frames to miss their deadlines. Worse, the resulting backlog may force the controller to skip subsequent frames to avoid further deadline violations. This places an upper bound on how aggressive probes can be. Counter to our intuition, the smaller the encoded frame size, the more aggressively the controller can probe without risking deadline misses. Syntra revealed that controllers must actively limit FEC aggressiveness to maintain latency bounds and that smaller frames enable larger probes—an insight that is easy to miss in heuristic designs [31].

❹ **Bandwidth probe ⇔ frame skipping** When a bandwidth probe reaches or exceeds the network's capacity, it induces packet delay and loss. Frame skipping becomes a critical tool for recovering from such events without violating latency bounds for all future frames. While reducing frame quality (i.e., lowering the target bitrate) is another option, it is less effective in practice—video encoders tend to respond slowly to target bitrate changes, making skipping the more responsive strategy. Syntra revealed that adaptive frame skipping is feasible as a recovery mechanism, where such certainty is difficult to obtain without rigorous analysis.

❺ **Bandwidth probe ⇔ frame bitrate** When the encoded frame exceeds what can be safely transmitted without loss, Syntra controllers cannot reliably infer network capacity unless the network exhibits no jitter during that interval. Such large frames may arise either from controller choices (e.g., requesting higher quality) or from encoder-side variability (e.g., keyframe insertion). Syntra showed that controllers should use FEC, rather than a larger bitrate, to probe for bandwidth.

❻ **Latency bound ⇔ past decisions** A frame's effective latency bound depends not only on the user-specified deadline but also on how long it has already been buffered at the sender.

This, in turn, is determined by earlier decisions about frame skipping, transmission timing, and bitrate. Overall, Syntra revealed that controllers must consider how their actions influence latency bounds for future frames and have a plan for meeting those bounds.

❼ **Bandwidth probe ⇔ latency bound** When probing for available bandwidth, the controller may delay sending frames in order to transmit them as a burst. After the burst, it may need to pause to let the queue drain before probing again. These delays consume part of the latency budget, reducing the effective bound for pending frames. Syntra showed that frame latency bounds must be aligned with bandwidth probes.

3.3 Conjectures about Fundamental Limits

Given assumptions about the environment and a user-specified objective, Syntra synthesizes both a controller and a corresponding performance bound. While these "guarantees" are not formally sound—the controller may not be optimal and the bound may be incorrect—only a few components of Syntra sacrifice rigor (see §5.6). Thus, it may be reasonable to conjecture that if Syntra fails to find a good controller under a given model, then no such controller exists within that model. Our experience supports this view, and we highlight two key insights that emerged from using Syntra.

Tighter latency budgets necessarily cause more frame skipping. Under the default system model (see §6), setting the latency bound to four propagation delays (i.e., one RTT in the absence of congestion) allows Syntra to synthesize a controller that consistently delivers high performance. The controller probes for bandwidth aggressively (using exponential increase) and skips frames only when a probe fails (i.e., when sending faster than the available capacity causes delay or loss) or when the video encoder produces a frame that exceeds the target bitrate. These events are relatively rare. However, when the latency bound is reduced to three propagation delays, the controller is forced to skip frames with every probe, regardless of the outcome. This behavior indicates a hard tradeoff: further tightening latency constraints comes at the direct cost of reduced frame rate.

Higher encoder variability demands larger end-to-end latency. When the encoder produces frames with highly variable sizes—such as during frequent keyframe insertion—the controller has less predictability in how much data needs to be transmitted within a fixed window. This variability exacerbates queuing delays, making it harder to satisfy hard latency bounds. Syntra-generated controllers often fail to find a feasible policy unless the end-to-end latency is increased to tolerate bursts of large frames. This indicates that encoder variability imposes a lower bound on achievable latency guarantees, even with optimal control.

4 Overview of Syntra

Figure 1 gives a high-level overview of Syntra's architecture for synthesizing controllers. The input to Syntra consists of (1) a symbolic model that describes how the network and video encoder behave (see §6.1), and (2) a declarative performance objective specifying the desired tradeoffs between latency and quality (see §6.2). Given these inputs, Syntra formulates the synthesis problem as a partially observable two-player game between the controller and the environment. The goal of synthesis is to find a winning strategy for the controller and express this strategy in a form that can be executed efficiently during deployment (as shown in Figure 2).

Game formulation. Syntra models the interaction between the controller and the environment as a *partially observable* two-player game. At each time step, the controller selects an action—such as choosing a target bitrate, a FEC budget, or whether to skip a frame—without access to the full system state. The environment, which includes the network and the video encoder, responds by transporting packets and assembling frames according to a set of latent variables (e.g., link rate and buffer size) that are not directly observable. Instead, the controller receives partial feedback, such as acknowledgments, encoded frame size, and detected losses. The interaction between the controller and the environment is formally specified using a symbolic model that defines constraints over both controller actions and latent environment behavior. The controller's objective is to satisfy a mix of safety and reachability conditions: it must avoid violating hard constraints like latency bounds (safety), while also achieving longer-term goals such as timely, high-quality delivery of frames (reachability). The resulting formulation is a classic partially observable game, in which the controller must plan under uncertainty by reasoning about how its actions influence the (unseen) state of the system.

Belief construction. Since the controller cannot observe internal variables such as link rate, buffer size, or queue length, it must reason about them indirectly. To do so, Syntra maintains a *belief set*, which captures all possible values these latent variables could take based on the controller's past actions and observations. This belief set serves as a compact summary of what the controller knows about the state of the system. Unlike prior approaches that rely on manual approximations [3], Syntra uses symbolic quantifier elimination to automatically derive tight bounds on each latent variable. These bounds are expressed as simple functions of the observable history and can be evaluated efficiently at runtime (see the Belief Construction component in Figure 2).

Belief-based game reformulation. To enable planning under uncertainty, Syntra reformulates the problem as a symbolic game in which the controller's state consists of its current observation, the belief set and a belief summary—a representation of the belief set simplified to contain only relevant

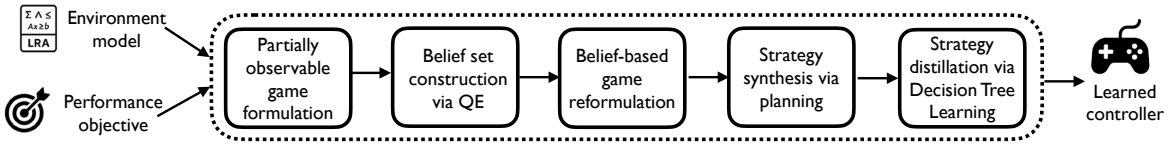


Figure 1: Illustration of Syntra's *offline* synthesis phase

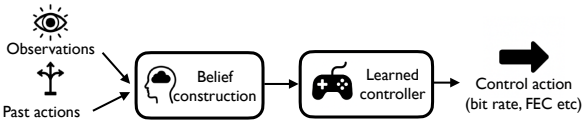


Figure 2: Illustration of Syntra's *online* deployment phase

information. This representation allows Syntra to reason about all admissible environment behaviors while enabling tractable planning using symbolic search techniques.

Minimax planning. Given a belief summary and observation, Syntra uses bounded-horizon minimax search to determine control actions that perform well under worst-case environment behavior. At each decision point, the controller selects an action to maximize the objective, while the environment adversarially selects an observation—constrained by the model—to minimize it. The planner explores all such action-observation sequences up to a fixed depth, evaluating leaf outcomes based on the specified performance goals. To keep this search tractable, Syntra incorporates pruning and search-ordering techniques adapted from game-playing AI (e.g., Chess engines). The resulting state-action pairs represent optimal behavior that can be used for training the final controller.

Decision tree learning. The planning approach described above is far too computationally expensive for use during deployment, as it requires unrolling the game tree and evaluating all possible environment responses at each step. Instead, Syntra uses the planner offline to generate a dataset of belief summaries paired with optimal actions, and then distills this behavior into a compact decision tree. This enables the controller to run efficiently in real time without lookahead or long-term state tracking. Moreover, the learned policy is human-readable, making it easy to inspect, debug, and integrate into real-world systems.

Online deployment. At runtime, the Syntra controller operates using only a compact sliding window of recent observations and actions, from which it computes the current belief summary. This summary is then passed to the decision tree to select the next control action, as depicted in Figure 2. Because all planning is done offline and the belief functions are compiled into closed-form expressions, the controller incurs minimal computational overhead and can operate effectively in long-running video sessions.

5 Detailed Design of Synthesis Framework

This section provides more details about the synthesis pipeline illustrated in Figure 1.

5.1 Environment Model and Game Set-up

As stated earlier, Syntra models control as a two-player game between the controller and the environment, evolving over discrete time steps. At each round t, the controller selects an action $\mathcal{A}_t$, the environment updates its internal (latent) state to $\mathcal{L}_t$, and an observation O_t becomes visible to the controller. The controller's action $\mathcal{A}_t$ includes the selected bitrate, FEC level, and skip decision. The observation O_t includes the number of packets acknowledged and lost, and the encoded frame sizes. The latent state $\mathcal{L}_t$ includes internal quantities not visible to the controller, such as the bottleneck link rate, the size of the network buffer, and the current buffer occupancy at the decoder. These latent variables critically affect performance but are not directly measurable.

System dynamics are defined by a symbolic environment model Model, written in linear rational arithmetic (LRA). The model encodes how the environment may evolve in response to controller actions, constrained by physical limits on encoding and transport. For example, it bounds how much actual frame size can vary from a target bitrate, or how many packets can be delivered based on available link capacity and queue occupancy. The model is expressed as an LRA formula:

$$\mathsf{Model}(\mathcal{A}_0^t, O_0^t, \mathcal{L}_0^t),$$

which holds if the trace of actions, observations, and latent states up to time t is valid according to the system's physical constraints. Here, X_0^t is shorthand for the set $\{X_0, \ldots, X_t\}$. We assume Model is prefix-closed: any valid trace implies all its prefixes are also valid.

As a quick example, consider a simplified network path model without jitter or buffering. We use the variable $A(t)$ to represent the number of bytes sent by the controller, $C(t)$ to denote the available link rate, and $S(t)$ to indicate the number of bytes acknowledged. For a single time-step transition, the model can be expressed as a piecewise-linear constraint in LRA over these variables:

$$(A(0) > C(0) \;\Rightarrow\; S(1) = S(0) + C(0))$$

$$\wedge\, (A(0) \leq C(0) \;\Rightarrow\; S(1) = S(0) + A(0)),$$

indicating that the number of acknowledged bytes at time 1 increases either by the full link rate (if the sender overshoots capacity) or by the number of bytes sent (if transmission remains within capacity).

Importantly, the environment is not required to commit to a fixed latent trajectory. At each round, it may choose any $\mathcal{L}_0^t$ that keeps the full trace consistent with Model. This includes varying quantities in the past—like the past buffer occupancy—so long as the resulting behavior remains valid. This flexibility enables generality but introduces epistemic uncertainty: from the controller's perspective, many latent sequences $\mathcal{L}_0^t$ may be consistent with the observed history. Rather than assuming a single hidden state, the controller must plan against all such possibilities.

Continuing the example above, suppose the controller sends $A(0) = 10$ bytes, and the acknowledgments change from $S(0) = 100$ to $S(1) = 110$. In this case, any link rate $C(0) \geq 10$ is consistent with the model, since the observed increase in acknowledgments could result from multiple possible link capacities.

5.2 Belief Computation

Because the controller does not observe the latent states $\mathcal{L}_0^t$ directly, it must reason about uncertainty over time. At each round t, the controller has access to a history of its own actions and the resulting observations:

$$H_t = (\mathcal{A}_0, O_0), (\mathcal{A}_1, O_1), \ldots, (\mathcal{A}_t, O_t).$$

Given this history, the controller must consider all latent state sequences $\mathcal{L}_0^t$ that are consistent with both the symbolic model Model and the observed behavior. We define the *belief set* as:

$$\mathsf{Belief}(H_t) = \{\mathcal{L}_0^t \mid \mathsf{Model}(\mathcal{A}_0^t, O_0^t, \mathcal{L}_0^t)\}.$$

This set represents the controller's epistemic uncertainty: all the latent state trajectories that could plausibly explain what it has seen so far. Because Model permits flexible latent evolution, this set may contain many sequences, even if the controller's observations are deterministic and repeatable.

Not all aspects of this set are necessary for the reformulation, so Syntra uses a small number of symbolic *belief summaries* to efficiently capture key aspects of this set. The summaries are closed-form functions derived from $\mathsf{Belief}(H_t)$ via Quantifier Elimination (QE). Specifically, for our video streaming domain, we use the following belief summaries:

- $B_C(H_t)$ is the tightest range of link rates C such that some trace in $\mathsf{Belief}(H_t)$ ends with $\texttt{link\_rate}(\mathcal{L}_t) = C$ (note, our model constrains C to be constant over time).
- $B_\beta(H_t, C)$ is the range of network buffer size β that are consistent with H_t, assuming the link rate is C.
- $B_q(H_t, C, \beta)$ gives the range of queue lengths q at round t, given link rate C and buffer size β.

Syntra computes these summaries offline by unrolling the symbolic model over a bounded time horizon and applying quantifier elimination. For example, to compute $B_C(H_t)$, it constructs a formula of the form:

$$\exists \mathcal{L}_0^t.\ \mathsf{Model}(\mathcal{A}_0^t, O_0^t, \mathcal{L}_0^t) \wedge \texttt{link\_rate}(\mathcal{L}_t) = C,$$

and then eliminates all latent variables $\mathcal{L}_0^t$ to obtain a closed-form LRA constraint over the history H_t. The resulting formula defines the feasible range of link rates consistent with H_t.

As a simple illustration, consider applying quantifier elimination to the network path example above:

$$\exists C(0), C(1).\ \mathsf{Model} \wedge C(0) = C \wedge C(1) = C.$$

Eliminating the latent variables $C(0)$ and $C(1)$ yields a closed-form LRA constraint over the observable history:

$$C \geq S(1) - S(0) \ \wedge\ (A(0) > S(1) - S(0) \ \Rightarrow\ C \leq A(0)).$$

From this result, we can clearly identify both a lower and an upper bound on the feasible link rate C, derived directly from the observations $\{(A(0), S(0)), (A(1), S(1))\}$. Specifically, the link rate must be at least the observed increase in acknowledged bytes and at most the number of bytes sent when the sender transmits beyond the available capacity.

Syntra uses a similar approach to determine the set of feasible next observations O_{t+1}. Given a history H_t and an action $\mathcal{A}_{t+1}$, Syntra constructs a formula asserting that there exists a latent sequence consistent with Model and the extended history $(H_t, (\mathcal{A}_{t+1}, O_{t+1}))$, then eliminates latent variables to characterize permissible observations symbolically.

Syntra performs quantifier elimination once per environment model and reuse the result thereafter. Most QE queries complete within a few hours; the largest required up to a week. Once computed, the resulting formulas are compiled into efficient Rust code and evaluated at runtime over a sliding window of recent actions and observations. These belief summaries define the controller's effective state in the reformulated game and serve as input to both bounded-horizon planning and the final decision tree policy.

5.3 Belief-Based Game Reformulation

The belief summaries introduced in the previous section allow Syntra to reformulate the problem into a belief-based game over observable states and beliefs. At each round t, the controller observes a tuple $(O_t, \mathcal{B}_t)$, where O_t is the current observation (e.g., ACKs, quality), and $\mathcal{B}_t = (B_C(H_t), B_\beta(H_t, C), B_q(H_t, C, \beta))$ summarizes the controller's uncertainty about the latent state, given the history H_t. The controller selects an action $\mathcal{A}_t$ based on this tuple. The environment then produces a new observation O_{t+1} that is consistent with the symbolic model Model and the current belief. The observation history is extended to $H_{t+1} =$

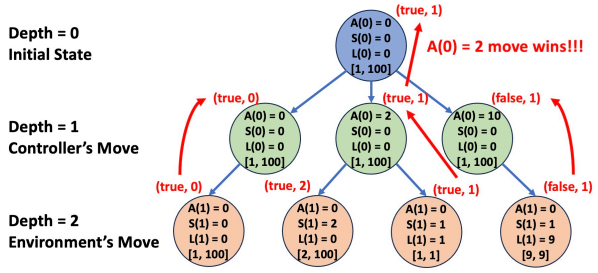


Figure 3: Example of the belief-based minimax search unfolded to depth 2.

$H_t \cup \{(\mathcal{A}_{t+1}, O_{t+1})\}$, and new belief summaries are computed. This reformulation enables planning over a compact state space defined by observation and belief pairs.

5.4 Bounded-Horizon Planning via Minimax

Given the belief-based game over $(O_t, \mathcal{B}_t)$ pairs, Syntra performs offline planning to identify optimal actions at each reachable state. The goal is to synthesize a strategy that satisfies application-level objectives—such as delivering high-quality frames on time—while anticipating worst-case environment behavior consistent with the symbolic model.

Syntra performs planning via a depth-d minimax search over the reformulated game. At each controller node, an action $\mathcal{A}_t$ is chosen; at each environment node, the adversary selects a valid next observation O_{t+1} consistent with the current belief and symbolic model Model. The history is extended to include the new action and observation, and updated belief summaries are computed symbolically for the next step.

Syntra unfolds the belief-based game tree up to a fixed depth d (12 in our implementation, corresponding to 6 moves each by the controller and environment) and evaluates the control objective at each leaf node. Payoffs are then backpropagated using standard minimax rules: at controller nodes, the action that maximizes the objective is chosen; at environment nodes, the worst-case (minimizing) observation is assumed. Because backpropagation compares only the scalar payoffs at leaves, the user-specified objective needs only to induce a total ordering over outcomes. The action selected at the root corresponds to the optimal choice given the current observation and belief.

Figure 3 shows a simplified example of Syntra's belief-based minimax search unfolded to two rounds: one controller move followed by one environment move. At each node, the controller maintains an interval $[\underline{C}, \overline{C}]$ representing its current bounds on the feasible link rate C, derived from the belief summaries. At depth 1, the controller chooses an action $A(0)$; at depth 2, the environment responds with an observation $(S(1), L(1))$—the cumulative acknowledged and lost bytes—and an updated belief interval consistent with the symbolic model Model. For illustration, suppose the objective is to keep packet loss below 2 and, subject to that, maximize acknowledgments. At the leaf nodes, this objective is evaluated numerically and propagated upward: controller nodes pick the action that maximizes the value, while environment nodes choose the worst-case observation that minimizes the value.

To improve scalability, Syntra applies standard alpha-beta pruning to eliminate subtrees that cannot affect the final decision. To make pruning more effective, it adopts three standard ideas from Chess engines: iterative deepening, transposition tables, and aspiration windows [15, 21]. Since belief summaries are symbolic functions compiled into efficient code, each node evaluation is fast and lightweight. The result is a labeled dataset of triples $(O_t, \mathcal{B}_t, \mathcal{A}_t)$, mapping each symbolic game state to the best action under bounded-horizon worst-case planning. Each triple is generated in under a minute and requires exploring between 10^4 and 10^5 nodes.

5.5 Decision Tree Learning

The planning procedure described above produces a dataset of labeled examples, each consisting of an observable state o_t, a belief summary β_t, and an optimal action a_t selected via minimax search. These examples are used to train a decision tree classifier based on standard greedy splitting criteria (information gain). The input to the tree includes both the current observable state and the components of the belief summary: for example, recent ACK counts and quality feedback from o_t, along with symbolic bounds on link rate, buffer size, and queue length from β_t. The output is a control action, such as a target bitrate, FEC configuration, or frame skipping decision.

We use a decision tree instead of a neural network for simplicity, transparency and efficiency. Each branch in the tree corresponds to clear, interpretable conditions over observable and belief features, making it easy to inspect, debug, and adjust the policy when needed. Tree evaluation is also extremely fast, requiring only a few conditional checks, which allows real-time deployment even on limited hardware. In contrast, neural networks are harder to interpret and verify, and their higher computational cost makes them less suitable for tight control loops.

At deployment time, Syntra evaluates the decision tree once per RTT to determine the control action. Since both belief computation and tree evaluation are lightweight, the controller introduces negligible runtime overhead. In our implementation, the full decision logic takes under a millisecond per decision on a single CPU core, making it suitable for real-time deployment even on resource-constrained devices.

5.6 Scalability and Optimality Tradeoffs

Syntra prioritizes scalability and practicality ove formal guarantees. In this section, we discuss the three sources of approximation that cause Syntra to lack either soundness or optimality guarantees. However, Syntra works very well in practice despite the lack of formal guarantees (see §7).

Discretization. First, because Syntra uses the Minimax algorithm for planning, it can only work for finite action spaces. As discussed in the next section, we choose a suitable discretization of the action space for the video streaming setting and manually refine our discretization strategy to achieve a good trade-off between scalability and quality.

Finite-horizon planning. Second, Syntra performs planning over a finite time horizon. As a result, the actions suggested by the controller may not be truly optimal. However, we believe that Syntra's use of belief summaries helps mitigate long-term dependencies. In particular, the computation of belief summaries from finite histories facilitates generalization, allowing the learned controller to indirectly account for past system behavior without considering arbitrarily long traces.

Imitation learning. Third, Syntra synthesizes the final policy by learning a decision tree based on a finite data set and does not formally verify the learned policy. However, empirically, we find that the resulting decision tree is simple, interpretable, and achieves 100% accuracy on the training data.

6 Instantiating Syntra for Video Streaming

While Syntra's synthesis framework (described in the previous section) can, in principle, be used to synthesize different types of controllers, the focus of this paper is learning a video streaming controller. In this section, we provide more details about how to instantiate Syntra's learning framework in the video streaming setting.

6.1 Environment Model

Modeling philosophy. We model the environment—comprising both the network and video encoder—as a non-stochastic but non-deterministic object. This abstraction captures a wide range of possible system behaviors using a compact and expressive mathematical formulation. To enable symbolic reasoning, they are specified as first order Linear Rational Arithmetic (LRA) formulas (see §5.1).

While non-deterministic models are often considered overly conservative—accounting for worst-case rather than typical behaviors—we find that this conservatism does not impede performance in practice. In fact, Syntra outperforms baseline approaches even under idealized conditions, such as jitter-free, fixed-rate links with deep buffers, despite being optimized for worst-case scenarios. The performance advantage only grows under more challenging real-world network conditions. These results suggest that, with appropriate structural assumptions, non-deterministic models can avoid excessive pessimism while still delivering strong generalization and performance.

In contrast, stochastic models require the modeler to specify precise probability distributions, which often necessitate assumptions that are difficult to justify. For example, even

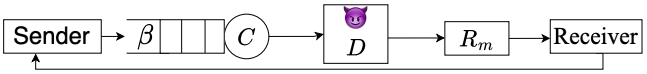


Figure 4: Network model

analyses that permit arbitrary distributions frequently assume variables are independently and identically distributed (i.i.d.), an assumption that rarely holds in the real world. The non-deterministic approach, by comparison, enables the modeler to assert only those assumptions needed to prove desired properties, leaving all other aspects unspecified.

Video encoder model. Given a target rate of x bytes per frame, we assume that the encoder produces a frame whose size lies within the range $[x/2, x]$, but no particular probability distribution is assumed within this interval. Empirically, we find that this bound holds for most frames across a range of video content for our encoder, though the specific constants depend on factors such as the encoder configuration and the number of frames transmitted per RTT. Syntra can synthesize controllers for any specified bounds, including looser intervals that allow frame sizes to exceed x. Importantly, the synthesized controller remains effective even when occasional violations of the bound occur—for example, during keyframe generation, as demonstrated in our evaluation (see §7.3).

In addition to frame size variability, rapid changes in the target bitrate can significantly degrade video quality, as encoders lose the ability to reuse blocks across frames. Rather than explicitly model this degradation, we impose a constraint: the controller may only adjust the target rate by at most a factor of $2\times$ relative to the previous value. This simple constraint improves quality, frame rate, and latency compared to baseline controllers that allow unconstrained bitrate swings (see §7).

Network model. We adapt a network model originally developed for verifying and synthesizing CCAs [3, 5] to the context of video streaming. As depicted in Fig. 4, the model represents the network path as a single bottleneck link with a constant, but unknown, capacity C (in bytes/s) and buffer size β (in bytes).

The model accounts for a wide range of real-world behaviors—including packetization effects, ACK aggregation, token-bucket shaping, and OS-level jitter—by allowing an adversarial process to delay packets arbitrarily, subject to two constraints: packets cannot be reordered, and no packet may be delayed by more than D seconds (we use $D = R_m$). This abstraction enables the model to emulate a broad class of practical network paths [3, 5] while remaining analytically tractable. Capturing such phenomena is critical: several past CCA designs have failed precisely because they ignored these edge effects [4, 5]. The no-reordering assumption is realistic in practice, as many transport protocols already tolerate bounded reordering (typically up to three packets) to support accurate loss detection [7, 29]; our formal model incorporates this 3-packet threshold explicitly.

Handling time-varying model parameters. As mentioned

earlier, the network model allows link rate (C) and buffer size (β) to vary over time. To accommodate such time-varying parameters, Syntra modifies belief computation to prioritize recent observations in two ways. First, if the current sequence of observations cannot be explained by any fixed choice of model parameters—that is, if the belief set becomes empty—Syntra recovers by recomputing beliefs over the longest recent suffix of the observation history that yields a non-empty set. This reflects the intuition that recent behavior is more likely to reflect current conditions than older data. Second, even if the belief set remains non-empty, stale observations can cause inferred bounds to become overly loose. For example, if the link rate increases but the controller has not yet sent at a high enough rate to observe it, past observations may understate available capacity. To mitigate this, we limit belief computation to the last 6 RTTs. Empirically, we find that this window yields sufficiently tight bounds on C and β, and that extending it further offers only marginal benefit.

When environment model assumptions are violated. While Syntra's synthesis procedure is designed to be robust under the assumptions encoded in the system model, we find empirically that the synthesized controller performs well even when these assumptions are violated—for example, when actual frame sizes exceed the modeled bound or when link capacity fluctuates more rapidly than assumed.

6.2 Performance Objective

Due to its use of minimax search, Syntra provides a flexible interface for specifying the objective: a black-box function that compares two sequences of actions and observations and returns which one is preferred. While traditional approaches summarize tradeoffs between competing goals as a single real-valued score, Syntra supports more expressive and interpretable objectives. In particular, it allows *lexicographic objectives*, where each sub-objective can be either a boolean property (e.g., safety or reachability) or a quantitative metric that encodes optimality. This structure enables users to express complex multi-stage preferences, such as prioritizing safety constraints before optimizing performance. We experimented with several objective functions and ultimately selected one whose behavior aligned best with our design goals. The components of the objective, listed below in lexicographic order, reflect increasingly lower priorities—the comparison between two traces depends only on the first point at which they differ:

1. Never lose frames,
2. If bandwidth probing has converged (i.e., $\max(B_C(H_t)) < 2\min(B_C(H_t))$), ensure every frame is delivered at most $2R_m$ after it has been encoded, else ensure it is delivered $4R_m$ after it has been encoded
3. Minimize the number of skipped frames,
4. Minimize the upper bound on C in the belief set,
5. Maximize the lower bound on C in the belief set.
6. Maximize the average frame quality,

At the top of the priority list is ensuring frame delivery: if a frame is lost after encoding, it stalls the decoder pipeline, making recovery difficult. To avoid this, we rank delivery above all other concerns. The next priority is meeting the latency bound. As discussed in § 3.3, the ability to meet latency constraints without frame skipping improves significantly once the controller's bandwidth estimate has stabilized. To reflect this, the objective uses a split condition: tighter latency requirements are imposed after convergence. The third component of the objective governs bandwidth probing. We prioritize minimizing the upper bound of the belief set before maximizing the lower bound, to encourage aggressive—but still safe—probing. However, probing near the upper bound after convergence increases the risk of frame skipping (see §3.2). To mitigate this, we slow the rate at which the upper bound estimate evolves: specifically, we define it as the minimum of $max(B_C(H_t))$ over the last second. This design limits steady-state probing to roughly once per second, while still enabling adaptation if capacity increases. Once a probe succeeds, the controller resumes regular probing until it reconverges. Finally, we include average frame quality and frame skipping rate as lower-priority objectives. The relative importance of these can be adjusted depending on the desired tradeoff: one could swap their positions in the ranking, or combine them into a weighted average to tune for frame rate versus visual fidelity.

Iterating on the objective. The performance objective used in our experiments was not designed upfront—it emerged through iteration. Real-time video streaming involves competing goals: quality, latency, reliability, encoding artifacts, and interactions with network dynamics. Capturing these tradeoffs in a single declarative objective is difficult, and we refined ours through many rounds of experimentation. Syntra enabled this process by making iteration over objectives fast: controller synthesis completes in 3–4 hours on a single CPU core,[1] and scales well with parallelism. In fact, the user can get insight into the controller's behavior by directly inspecting the generated actions, even before enough data has been computed for imitation learning. This rapid turnaround allowed us to explore a wide range of objectives and converge on a formulation that produced high-quality behavior.

6.3 Discretization of the Action Space

Syntra requires the user to discretize both the controller and environment action spaces. In the context of video streaming, we refined these discretizations to balance quality and planning efficiency.

Environment actions. The environment's actions include the number of packet acknowledgments, packet losses, and

[1] Excluding quantifier elimination, which is a one-time preprocessing step. We have not had to re-run it in over a year.

the size of encoded frames. Initially, we discretized each dimension into four values, yielding 64 possible environment actions. However, empirical analysis revealed that the planner consistently preferred extreme configurations (e.g., maximum or minimum loss and acknowledgment amounts). Based on this insight, we reduced the environment's discretization to eight "corner case" actions, significantly lowering the branching factor while preserving the expressiveness needed to model worst-case network behaviors.

Controller actions. The controller must make decisions about sending rate, FEC redundancy, frame grouping for FEC, frame skipping, and target encoding bitrate. We began with a uniform discretization of each control dimension. However, as we iterated, it became clear that some control strategies required values not aligned with a uniform grid. In particular, actions such as sending at a rate of $C_{\text{min}} - \beta_{\text{min}} - q_{\text{max}}$—where C_{min}, β_{min}, and q_{max} are derived from the current belief summaries—emerged as critical for safely maximizing throughput. As a result, we supplemented the uniform grid with a small number of semantically meaningful action values derived from belief-based reasoning. This hybrid discretization improved controller expressiveness without significantly impacting planning efficiency.

Reducing redundancy. Frame grouping for FEC introduces additional complexity: given the other four control choices, there is a unique way to group frames that maximizes FEC protection without violating latency bounds. We derived this mapping analytically and eliminated redundant grouping options from the search space. This pruning further reduced the planner's branching factor, enabling deeper search.

7 Evaluation

We evaluate Syntra on real-time video streaming over lossy networks, comparing its performance, robustness, and design components against strong baselines.

7.1 Setup

Testbed. We built a C++-based testbed for headless peer-to-peer video streaming, extending the core architecture of Salsify [20]. The sender reads video frames from an input Y4M file, while the receiver writes the output to a separate Y4M file. Each frame is annotated with a unique 2D barcode to enable frame-level matching for visual quality and latency evaluation. To emulate variable network conditions, we place the sender behind a Mahimahi link shell, allowing controlled manipulation of bandwidth and delay. Unless otherwise specified, all experiments use a lossless link with a fixed round trip propagation delay of 50 ms, and each run lasts 10 minutes.
Metrics. We use three primary metrics to compare different techniques: frame latency, frame quality, and received frame rate. Frame latency is measured end-to-end: that is, the time between when the frame was read at sender and when it was decoded. Frame quality is measured by the Structural Similarity Index Measure (SSIM).
Network Traces. We evaluate each scheme on a set of six real-world, 10-minute-long cellular network traces [46]. To further stress the controllers under controlled conditions, we also use synthetic traces that vary along several dimensions: fixed versus time-varying link capacities, shallow versus deep network buffers, and the presence or absence of ACK aggregation.
Videos. We use a 720P video with a frame rate of 30 FPS Y4M video downloaded from YouTube. Audio is disabled during the experiments.
Baselines. We evaluate our approach against three joint-designing baselines: (1) WebRTC [1] with Google Congestion Control (GCC) [11], a widely used standard that serves as a strong reference point for real-time communication systems; (2) Salsify [20], a prior state-of-the-art system that jointly controls video encoding and congestion for low-latency video streaming; and (3) WebRTC with GCC replaced by Vegas [8], a delay-based congestion control algorithm that adjusts the sending rate based on queuing delay rather than loss. Note, Salsify's video encoder is limited to 20 FPS. In experiments where we do not match/outperform it on other metrics we also run Syntra on 20 FPS video for a fair comparison.

7.2 Overall Comparison.

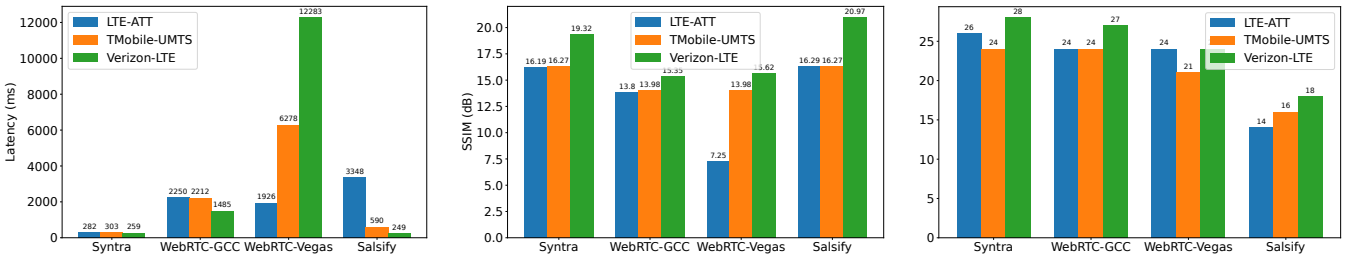
We compare Syntra against WebRTC-GCC, WebRTC-Vegas, and Salsify using the real-world cellular traces described in Section 7.1. Experiments are conducted on Mahimahi [37] trace replay, with a fixed round trip propagation delay of 50 ms and either infinite[2] or shallow (0.5 BDP) buffering at the receiver. Each run lasts for 10 minutes. The T-Mobile UMTS and Verizon-LTE traces are particularly challenging, with available bandwidth oscillating between 20 Mbps and 100 kbps over short timescales.

The results of the evaluation are summarized in Figure 5. Across all traces and metrics—tail latency, video quality, and frame delivery rate—Syntra consistently outperforms all baselines other than Salsify, which has a much lower frame rate.

P95 latency. Subfigure (a) shows the P95 one-way delay achieved by each scheme. Syntra achieves more than a 6× reduction in P95 latency compared to WebRTC-GCC and WebRTC-Vegas, and a substantial improvement compared to Salsify. These gains hold under both infinite and shallow buffering configurations.

Video quality. Subfigure (b) reports the video quality measured by SSIM. Syntra improves median SSIM by more than 2 dB relative to WebRTC across all traces. Quality improvements are consistent even under high variability and loss, and

[2]This is for the benefit of the baselines, since all schemes are resistant to buffer bloat and Syntra handily outperforms baselines on shallow buffers



(a) P95 Latency (ms) **(b)** Median Quality (SSIM dB) **(c)** Average Frame Rate (fps)

Figure 5: The Comparison of Syntra, WebRTC-GCC, WebRTC-Vegas, and Salsify over three real-world network traces (AT&T, T-Mobile, Verizon). The metrics include P95 Latency, Median Quality, and Average Frame Rate.

Syntra maintains better visual quality throughout.

Frame delivery rate. Subfigure (c) shows the fraction of frames successfully delivered. Syntra consistently delivers a higher percentage of frames than WebRTC-GCC, WebRTC-Vegas, and Salsify. This advantage is maintained across both real traces and different buffering settings.

Summary. These results demonstrate that Syntra achieves lower latency, higher video quality, and better frame continuity compared to baseline approaches across a variety of cellular network conditions.

7.3 Understanding the Controller

Trace representation. Figure 6 shows the behavior of a Syntra-generated controller streaming real video over an emulated 3 Mbps link (14 MTU-sized packets per R_m). In (a), the link has no delay jitter; in (b), we introduce 50 ms ACK aggregation every 50 ms. The controller makes decisions every R_m seconds ($R_m = 50$ ms here), matching the minimum feedback delay. Each horizontal bar represents a video frame, spanning from encoding to full transmission. Since frames are generated every 33 ms, not aligned with R_m, some timesteps include one frame while others include two. Frames are grouped into batches according to their delivery deadlines. A frame generated at time t must be received by $t + 4R_m$ (200 ms) to meet its latency target. Transmission can only begin after encoding completes. In an ideal case without queuing, packets arrive within R_m to $R_m + D$ (1–2 RTTs); otherwise, queueing increases latency. Vertical bars show the number of packets transmitted per batch, separating original and FEC packets.

We next describe key behaviors observed, referencing the numbered insights from §3.2 (❶–❼).

Types of probing. The probing behavior of the Syntra-generated controller adapts dynamically to the situation. In Fig. 6a, time steps 3, 5 and 7 illustrate the ideal probing scenario: frames have large latency bounds, allowing them to be batched (❼) and encoded with joint FEC (❶), and the absence of network jitter enables the controller to infer that the queue is empty. As a result it can add significant FEC redundancy without risking the latency of subsequent frames (❸). At time step 9, the probe initially resembles an ideal probe but then encounters an upper bound on C, and $C_{\max}$ reduces from infinity. Consequently, the controller must skip frames to recover (❹).

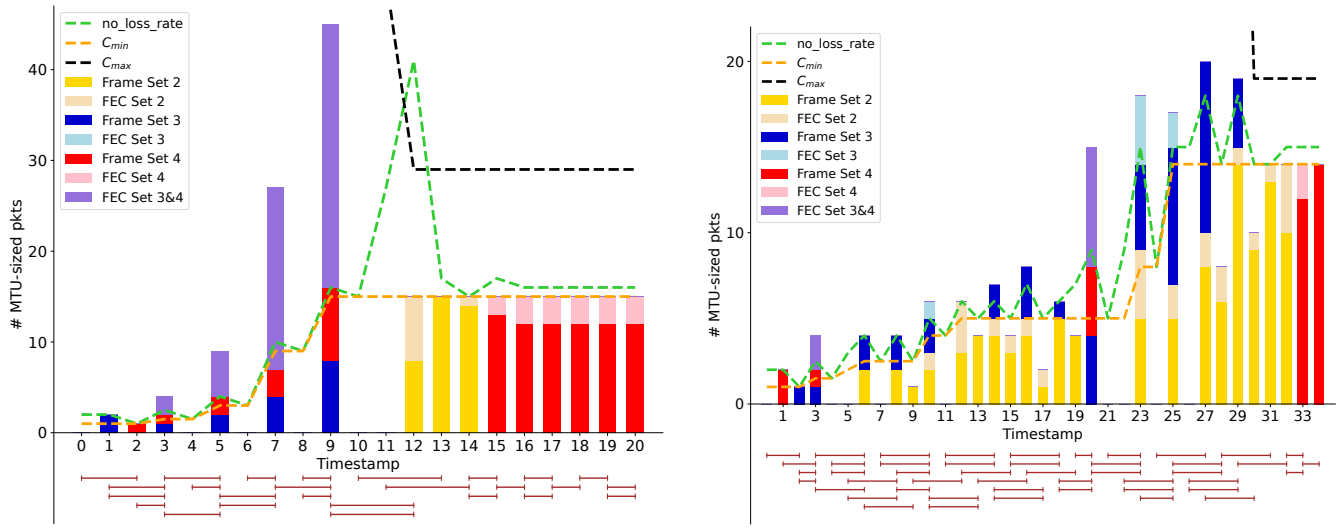
Model violation. At time step 1 in Fig. 6a and time steps 1, 14 and 16 in Fig. 6b, a different challenge arises: the encoder generates a frame that exceeds the expected size. The Syntra-generated controller responds gracefully by spreading the frame's transmission across two time steps to avoid frame loss (❷). Recognizing that probing for bandwidth is risky under such conditions, it abstains from adding FEC redundancy (❺) and $C_{\min}$ does not increase.

FEC grouping with jitter. In Fig. 6b, Syntra only applies FEC over a single frame set (❶). This is because the latency bounds are too tight for joint FEC (❻).

Probing with jitter. In the presence of jitter, the controller adopts a more robust but slower strategy to increase $C_{\min}$. Rather than increase $C_{\min}$ directly, it first increases the lower bound on $C + \beta$, which yields a higher bound on `no_loss_rate`. To do so, it must spread the probe across two time steps since the latency bounds are too tight for joint FEC (❶). Now that `no_loss_rate` is higher at time step 20, it can be more aggressive in probing for $C_{\min}$ at timestep 23. This is a complex compound strategy that Syntra discovered unexpectedly.

7.4 Evaluation on Microbenchmarks

To better understand the behavior of the synthesized controller, we evaluate Syntra under targeted synthetic scenarios that stress specific aspects of the network environment. We focus on three challenges: (1) rapid response to variation in link capacity, (2) resilience to periodic ACK aggregation, and (3) avoiding quality loss under shallow buffering. These experiments isolate key stressors that arise in real-world conditions and allow finer-grained comparisons against baseline systems. In all cases, we use a Mahimahi link emulator to impose controlled network conditions. Unless otherwise specified, the link has a fixed round-trip delay of 50 ms and the experiments



(a) Path without delay jitter

(b) Path with 50 ms of ACK aggregation

Figure 6: Decisions taken by a synthesized controller when streaming a real video over emulated links with a fixed rate (3 Mbit/s) and propagation delay (50 ms)

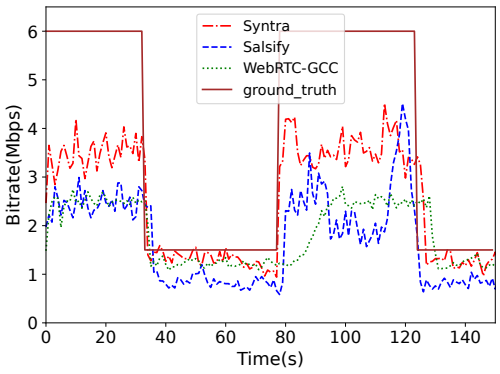


Figure 7: Average utilization of Syntra, Salsify and WebRTC-GCC on a video stream.

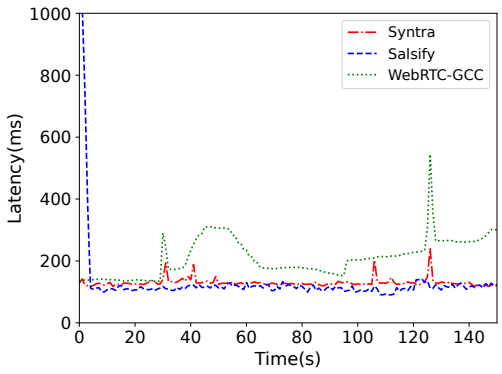


Figure 8: Average latency of Syntra, Salsify and WebRTC-GCC on a video stream.

last for 10 minutes. We compare Syntra to WebRTC-GCC, Salsify, and their variants under matched settings in Table 1.

Varying link rate. In this experiment, we evaluate how each tool adapts to periodic bandwidth shifts between 1.5 Mbps and 6 Mbps. We run the experiment for 2.5 minutes. Fig. 7 shows the actual video bitrate produced by each system (i.e., throughput minus FEC). All three tools converge in roughly the same time when the link rate decreases. However, under rising bandwidth conditions, Syntra converges noticeably faster than the others, with WebRTC-GCC exhibiting the slowest ramp-up. More importantly, across both downward and upward transitions, our system tracks the actual link rate with the highest fidelity. This responsiveness and accuracy underlie its advantage over baselines on real cellular traces.

Next, we compare end-to-end latency under fluctuating link conditions. In Fig. 8, WebRTC-GCC shows the weakest tolerance: after the rate drops at 37 s, its latency takes roughly 50 s to stabilize. In median latency, Salsify outperforms Syntra (111 ms vs. 125 ms) but at the 95th percentile, Syntra is better (154 ms vs. 218 ms). Furthermore, after revising the video's frame rate from 30 fps to 20 fps, Syntra achieves the same median latency (125 ms) as Salsify.

ACK Aggregation. In this experiment, we measure each tool's robustness under periodic ACK aggregation. We emulate a 3 Mbps link with 50 ms round-trip delay and impose 50 ms aggregation bursts every 50 ms. Only Syntra is able to handle these bursts well: its median latency and P95 latency both increase by approximately 25 ms. This is unavoidable given the imposed 50 ms jitter. In contrast, WebRTC-GCC fares poorly under ACK aggregation, with its median latency

System	Micro Trace	Video Quality (SSIM dB)		Video Delay (ms)		Frame Rate (fps)
		p25	median	median	p95	
Syntra	Fixed Link Rate	18.3	18.8	130	153	30
WebRTC-gcc		15.6	15.7	179	207	30
Syntra-20fps		19.0	19.6	119	146	20
Salsify		19.6	20.9	121	229	20
Syntra	Varying Link Rate	17.3	18.9	125	154	29
WebRTC-gcc		15.3	15.5	249	409	29
Syntra-20fps		19.5	20.4	111	144	20
Salsify		18.2	20.7	111	218	20
Syntra	ACK Aggregation	18.2	18.5	151	185	28
WebRTC-gcc		15.5	15.7	233	297	29
Syntra-20fps		19.2	19.8	134	181	20
Salsify		18.9	20.7	150	323	19
Syntra	Shallow Buffer	18.3	18.6	132	154	30
WebRTC-gcc		15.2	15.4	125	160	29
Salsify		15.2	15.5	115	160	17

Table 1: Performance comparison of Syntra and baselines across different network conditions (lower is better). Metrics are reported at the 25th percentile, median, and 95th percentile where appropriate. Syntra-20fps is supplied with 20 FPS video for fair comparison with Salsify

rising from 179 ms to 233 ms (an increase of 54 ms) and its P95 latency increasing from 207 ms to 297 ms, indicating low tolerance for aggregation effects. For Salsify, P95 latency increases from 229 ms to 323 ms, also showing limited robustness. In addition, running Syntra at 20 fps yields a $\sim$0.3 dB improvement in SSIM compared to Salsify.

Shallow buffer. In our final experiment, we assess each tool's resilience under shallow-buffer conditions. We emulate a 3 Mbps link with 50 ms round-trip delay and cap the buffer at 9,375 bytes—half of the bandwidth–RTT product. Under this constraint, Salsify's video quality falls by roughly 3 dB at both the 25th-percentile and median SSIM, whereas Syntra maintains the same quality as in the infinite-buffer case. In terms of frame rate, Salsify suffers the largest drop ($\sim$15%) due to packet loss and frame skips, while Syntra and WebRTC-GCC each incur a decrease of at most 1%. WebRTC-GCC's behavior is expected, as it typically behaves conservatively and underutilizes the available bandwidth.

Fixed Rate. We also compare performance in the easiest setting: fixed link capacity with a deep buffer. This shows that Syntra's gains come not only from its robustness to an adversarial model, but also its ability to exploit opportunities in joint control.

Summary. Overall, these targeted experiments show that Syntra not only improves quality and delay under typical conditions but also remains robust under stressors such as bandwidth collapse, ACK aggregation, and limited buffering capacity.

8 Conclusion

This paper presents Syntra, a system that synthesizes cross-layer video streaming controllers from symbolic models and declarative performance objectives. Syntra formulates control as a partially observable game and uses bounded-horizon minimax planning, along with decision tree learning, to generate robust controllers that adapt to uncertain and dynamic network conditions. Unlike prior approaches that rely on hand-tuned heuristics or learned policies from simulation, Syntra enables principled design exploration without requiring data collection or extensive manual tuning. Evaluation across real-world and synthetic network traces shows that Syntra consistently improves video quality, latency, and frame delivery rates compared to strong baselines, uncovering adaptation strategies that are difficult to engineer manually.

Acknowledgments

We would like to thank anonymous revierwers and our shepherd Sanjay Rao for feedback that helped improve our paper and Xiao Zhang for helping prepare the baseline. This work was conducted in research groups supported by NSF awards CNS-2403026, CCF-2422130, CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF-2210831, CCF-2319471, CCF-2422130, and CCF-2403211 as well as a DARPA award under agreement HR00112590133.

In memory of Prateesh Goyal

References

[1] World Wide Web Consortium (W3C). *WebRTC 1.0: Real-time Communication Between Browsers*. https://www.w3.org/TR/webrtc/. W3C Recommendation. 2021.

[2] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. "Classic meets modern: A pragmatic learning-based congestion control for the internet". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 632–647.

[3] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. "Towards provably performant congestion control". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 951–978. URL: https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. "Starvation in End-to-End Congestion Control". In: *Proceedings of the 2022 ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 2022.

[5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. "Toward Formally Verifying Congestion Control Behavior". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1–16. URL: https://doi.org/10.1145/3452296.3472912.

[6] Venkat Arun and Hari Balakrishnan. "Copa: Practical Delay-Based Congestion Control for the Internet". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 329–342. URL: https://www.usenix.org/conference/nsdi18/presentation/arun.

[7] Jon CR Bennett, Craig Partridge, and Nicholas Shectman. "Packet reordering is not pathological network behavior". In: *IEEE/ACM Transactions on networking* 7.6 (1999), pp. 789–798.

[8] L.S. Brakmo and L.L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (1995), pp. 1465–1480.

[9] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. "How I Learned to Stop Worrying About CCA Contention". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 229–237.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53. URL: http://queue.acm.org/detail.cfm?id=3022184.

[11] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. "Analysis and design of the google congestion control for web real-time communication (WebRTC)". In: *Proceedings of the 7th International Conference on Multimedia Systems*. 2016, pp. 1–12.

[12] Ke Chen, Han Wang, Shuwen Fang, Xiaotian Li, Minghao Ye, and H Jonathan Chao. "RL-AFEC: adaptive forward error correction for real-time video communication based on reinforcement learning". In: *Proceedings of the 13th ACM Multimedia Systems Conference*. 2022, pp. 96–108.

[13] Sheng Cheng, Han Hu, and Xinggong Zhang. "Abrf: Adaptive bitrate-fec joint control for real-time video streaming". In: *IEEE Transactions on Circuits and Systems for Video Technology* 33.9 (2023), pp. 5212–5226.

[14] David D Clark, Steven Bauer, William Lehr, KC Claffy, Amogh D Dhamdhere, Bradley Huffaker, and Matthew Luckie. "Measurement and analysis of Internet interconnection and congestion". In: *2014 TPRC Conference Paper*. 2014.

[15] Chessprogramming Wiki contributors. *Aspiration Windows and Iterative Deepening*. https://www.chessprogramming.org/Aspiration_Windows and https://www.chessprogramming.org/Iterative_Deepening. Accessed: 2025-04-14. 2025.

[16] Rayna Dimitrova. "Synthesis and control of infinite-state systems with partial observability". In: (2013).

[17] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. "PCC: Re-architecting congestion control for consistent high performance". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 395–408.

[18] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. "PCC Vivace: Online-Learning Congestion Control". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 343–356.

[19] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. "Equation-based congestion control for unicast applications". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '00. Stockholm, Sweden: Association for Computing Machinery, 2000, pp. 43–56. URL: https://doi.org/10.1145/347059.347397.

[20] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. "Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 267–282. URL: https://www.usenix.org/conference/nsdi18/presentation/fouladi.

[21] Colin Frayn. "Computer chess programming theory". In: *Retrieved October* 28 (2005), p. 2009.

[22] Daniel Genin and Jolene Splett. "Where in the Internet is congestion?" In: *arXiv preprint arXiv:1307.3696* (2013).

[23] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. "Elasticity Detection: A Building Block for Internet Congestion Control". In: *arXiv* (Feb. 2018). eprint: 1802.08730.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. URL: https://doi.org/10.1145/1400097.1400105.

[25] Chun-Ying Huang, Kuan-Ta Chen, De-Yu Chen, Hwai-Jung Hsu, and Cheng-Hsin Hsu. "GamingAnywhere: The first open source cloud gaming system". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 10.1s (2014), pp. 1–25.

[26] Tianchi Huang, Rui-Xiao Zhang, Chao Zhou, and Lifeng Sun. "QARC: Video quality aware rate control for real-time video streaming based on deep reinforcement learning". In: *Proceedings of the 26th ACM international conference on Multimedia*. 2018, pp. 1208–1216.

[27] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. "A buffer-based approach to rate adaptation: Evidence from a large video streaming service". In: *Proceedings of the 2014 ACM conference on SIGCOMM*. 2014, pp. 187–198.

[28] Marco Iorio, Fulvio Risso, and Claudio Casetti. "When latency matters: measurements and lessons learned". In: *ACM SIGCOMM Computer Communication Review* 51.4 (2021), pp. 2–13.

[29] Jana Iyengar and Ian Swett. "QUIC loss detection and congestion control". In: *RFC 9002* (2021).

[30] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. "A deep reinforcement learning perspective on internet congestion control". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3050–3059.

[31] Pantea Karimi, Sadjad Fouladi, Vibhaalakshmi Sivaraman, and Mohammad Alizadeh. "Vidaptive: Efficient and Responsive Rate Control for Real-Time Video on Variable Networks". In: *arXiv preprint arXiv:2309.16869* (2023).

[32] Taran Lynn, Dipak Ghosal, and Nathan Hanford. *Model Predictive Congestion Control for TCP Endpoints*. 2020. arXiv: 2002.09825 [cs.NI].

[33] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. "Neural Adaptive Video Streaming with Pensieve". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 197–210. URL: https://doi.org/10.1145/3098822.3098843.

[34] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. "PCC proteus: Scavenger transport and beyond". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 615–631.

[35] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. "Enabling high quality {Real-Time} communications with adaptive {Frame-Rate}". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 1429–1450.

[36] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. "End-to-end transport for video QoE fairness". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 408–423.

[37] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. "Mahimahi: Accurate Record-and-Replay for HTTP". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 417–429. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/netravali.

[38] Mirko Palmer, Malte Appel, Kevin Spiteri, Balakrishnan Chandrasekaran, Anja Feldmann, and Ramesh K. Sitaraman. "VOXEL: cross-layer optimization for video streaming with imperfect transmission". In: *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 359–374. URL: https://doi.org/10.1145/3485983.3494864.

[39] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of reactive (1) designs". In: *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings 7*. Springer. 2006, pp. 364–380.

[40] Amir Pnueli and Roni Rosner. "On the synthesis of a reactive module". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 179–190.

[41] *RFC 6582: The NewReno Modification to TCP's Fast Recovery Algorithm*. [Online; accessed 20. Nov. 2024]. Nov. 2024. URL: https://datatracker.ietf.org/doc/html/rfc6582 (visited on 11/20/2024).

[42] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. "BOLA: Near-optimal bitrate adaptation for online videos". In: *IEEE/ACM transactions on networking* 28.4 (2020), pp. 1698–1711.

[43] K. Tan, J. Song, Q. Zhang, and M. Sridharan. "A Compound TCP Approach for High-Speed and Long Distance Networks". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–12.

[44] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. "FAST TCP: Motivation, Architecture, Algorithms, Performance". In: *IEEE/ACM Transactions on Networking* 14.6 (2006), pp. 1246–1259.

[45] Keith Winstein and Hari Balakrishnan. "TCP Ex Machina: Computer-Generated Congestion Control". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 123–134. URL: https://doi.org/10.1145/2486001.2486020.

[46] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 459–471. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein.

[47] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. "Learning in situ: a randomized experiment in video streaming". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 495–511.

[48] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. "Pantheon: the training ground for Internet congestion-control research". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 731–743. URL: https://www.usenix.org/conference/atc18/presentation/yan-francis.

[49] Chenxi Yang, Divyanshu Saxena, Rohit Dwivedula, Kshiteej Mahajan, Swarat Chaudhuri, and Aditya Akella. "C3: Learning Congestion Controllers with Formal Certificates". In: *arXiv preprint arXiv:2412.10915* (2024).

[50] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. "A control-theoretic approach for dynamic adaptive video streaming over HTTP". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015, pp. 325–338.

[51] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. "Quality estimation models for gaming video streaming services using perceptual video quality dimensions". In: *Proceedings of the 11th ACM multimedia systems conference*. 2020, pp. 213–224.

A The Synthesized Joint Controller

Algorithm 1 Decide FEC Pattern

Require: Belief summaries of network states `bb`; metrics of the frame queue `frame_metrics`; small threshold α

Ensure: FEC pattern `pattern`

```
fec_pattern ← DRAIN
if bb.max_c ≤ 2× bb.min_c then
    pattern ← CONVERGED
else if bb.max_q ≤ α then
    d ← frame_metrics.get_deadline_in_RTT()
    if d ≤ 2 then
        fec_pattern ← PER_FRAME_SET_FEC
    else
        fec_pattern ← MULTI_FRAME_SET_FEC
    end if
end if
return pattern
```

Algorithm 2 Decide FEC Amount

Require: metrics of the frame queue `frame_metrics`; belief summaries `bb`; FEC pattern `fec_pattern`
Ensure: FEC amount `fec_amount`

```
fec_amount ← 0
set_size_1 ← frame_metrics.get_set(1).size()
set_size_2 ← frame_metrics.get_set(2).size()
if fec_pattern = MULTI_FRAME_SET_FEC then
    fec_amount ← 3 × bb.min_c − set_size_1 − set_size_2
else if fec_pattern = PER_FRAME_SET_FEC then
    latency_limit ← 3× bb.min_c − bb.no_loss_rate − set_size_2
    loss_limit ← bb.no_loss_rate − set_size_1 − set_size_2
    fec_ amount ← min(latency_limit, loss_limit)
else if fec_pattern = CONVERGED then
    fec_amount ← bb.min_c − set_size_1
else
    fec_amount ← 0
end if
return fec_amount
```

Algorithm 3 Decide Sending Rate

Require: metrics of the frame queue `frame_metrics`; belief summaries `bb`; FEC pattern `fec_pattern`; FEC amount `fec_amount`
Ensure: Sending Rate `rate`

```
fec_amount ← 0
set_size_1 ← frame_metrics.get_set(1).size()
set_size_2 ← frame_metrics.get_set(2).size()
if fec_pattern = MULTI_FRAME_SET_FEC then
    rate ← 3 × bb.min_c
else if fec_pattern = PER_FRAME_SET_FEC then
    rate ← bb.no_loss_rate + fec_amount
else if fec_pattern = CONVERGED then
    rate ← bb.min_c
else
    rate ← bb.min_c − bb.max_q
end if
return fec_amount
```

Algorithm 4 Decide Frame Skip

Require: Belief summaries of network states `bb`; metrics of the frame queue `frame_metrics`
Ensure: Whether to skip the next frame `frame_skip`

```
frame_skip ← False
if bb.max_c ≤ 2× bb.min_c then
    if d ≤ 2 then
        frame_skip ← True
    end if
end if
return pattern
```

Controller overview. The Syntra controller jointly decides the sending rate, bitrate, FEC redundancy, and frame-skipping behavior to satisfy RTT-bounded latency objectives. Algorithms 1–4 (Algorithms 1–4) present this logic as a sequence of structured conditional rules equivalent to the decision tree synthesized by imitation learning, but in a more readable form. At runtime, the controller takes as input the belief summaries `bb`—which provide lower and upper bounds on link capacity (`bb.min_c`, `bb.max_c`) and a conservative estimate of queue length (`bb.max_q`)—along with frame-queue metrics `frame_metrics`, including per-set size (a frame set comprises all frames generated within one RTT) and deadline (in RTTs).

FEC pattern and amount. Algorithm 1 (**Algorithm 1**) determines the FEC pattern. If the link has converged (`bb.max_c` $\leq 2\times$ `bb.min_c`), the controller selects the `CONVERGED` mode, entering a stable regime with minimal probing. Otherwise, if the queue remains below a small threshold α, the controller enters a probing phase. When the earliest deadline is tight ($d \leq 2$ RTTs), it chooses `PER_FRAME_SET_FEC`, encoding each set separately for fast recovery; otherwise, it switches to `MULTI_FRAME_SET_FEC`, merging multiple sets to increase redundancy and bandwidth utilization.

Algorithm 2 (**Algorithm 2**) computes the FEC amount corresponding to the chosen pattern. For `MULTI_FRAME_SET_FEC`, the redundancy equals the three-RTT envelope ($3\times$`bb.min_c`) minus the payload of the merged sets. For `PER_FRAME_SET_FEC`, the controller intersects the latency-limited and loss-limited budgets—derived from `bb.min_c`, `bb.no_loss_rate`, and frame sizes—to obtain a safe parity allocation. When `CONVERGED`, parity simply tops up the current set to `bb.min_c`.

Sending rate and frame skip. Algorithm 3 (**Algorithm 3**) translates these symbolic decisions into a concrete sending rate. Probing modes (`MULTI` or `PER_FRAME`) use higher rates proportional to the estimated capacity and FEC amount, while the converged mode stabilizes at `bb.min_c`. If the controller detects excess backlog, it lowers the rate accordingly to drain the queue. Algorithm 4 (**Algorithm 4**) introduces a lightweight frame-skipping safeguard: when the head-of-line frame's deadline is within one RTT or the total queued payload exceeds the safe budget under `bb.min_c`, the controller skips encoding the next frame to maintain the latency bound.