https://crad.ict.ac.cn
x(x): xx−xx, xxxx

基于传输层信息的视频码率自适应算法

陈方舟　王清楠　单丹枫
（西安交通大学计算机科学与技术学院　西安　710049）
（2193311601cfz@stu.xjtu.edu.cn）

Adaptive Video Streaming Based on Transport Layer Information

Chen Fangzhou, Wang Qingnan, and Shan Danfeng
（*School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an* 710049）

Abstract　ABR(adaptive bitrate) algorithms play a crucial role in enhancing the QoE(quality of experience) for online video streaming. Existing ABR algorithms rely on network characteristics observed at the application layer for bitrate decisions. However, this approach has inherent limitations: accurate video chunk download times cannot be fully derived from application layer observations. Specifically, these algorithms overlook critical factors such as RTT(round-trip time) and packet loss rate, which impact video chunk transmission, and they exhibit limited responsiveness to rapid network fluctuations. To address this, this paper propose Prophet, a bitrate adaptation algorithm based on transport layer information. Unlike traditional ABR algorithms, Prophet directly calculates network parameters such as bandwidth, packet loss rate, and RTT at the transport layer, enabling a more accurate assessment of network conditions. Additionally, a download time prediction model is developed based on transport-layer feedback, taking into account factors like packet loss retransmission and tail latency to achieve precise download time predictions. Experiments conducted in real-world network environments demonstrate that the Prophet algorithm performs well under various network conditions. Compared to existing ABR algorithms, Prophet achieves an average QoE improvement of 0.3%-117.9%, with a notable average QoE increase of 31.7%-117.9% in cellular network conditions.
Key words　video streaming transmission; bitrate adaptation; DASH; quality of experience; network round-trip time

摘　要　码率自适应（adaptive bitrate, ABR）是提升在线视频用户观看体验的重要方法。现有视频流码率自适应算法通常基于应用层观测到的网络特征来进行码率决策，但是这一方法存在明显局限：仅基于应用层观测难以准确估计出视频块下载时间。具体而言，现有算法往往忽视了往返时延、丢包率等因素对视频块传输的影响，且实时性较差。为此，提出了一种基于传输层信息的视频码率自适应算法 Prophet。与传统的 ABR 算法不同，Prophet 在传输层实时计算带宽、丢包率和往返时延等网络参数，从而更精确地评估网络环境。此外，还基于传输层信息建立了视频块下载时间预测模型，系统地考虑了丢包重传、尾部时延等因素，实现了对下载时间的精准预测。基于真实网络环境的实验表明，Prophet 算法在多种网络条件下均优于现有算法。与现有 ABR 算法相比，Prophet 的平均用户体验质量（quality of experience, QoE）提高了 0.3%~117.9%. 在蜂窝网络环境下，Prophet 的平均 QoE 提高了 31.7%~117.9%。
关键词　视频流传输；码率自适应；基于 HTTP 的动态自适应流；体验质量；网络往返时延

收稿日期：2024-11-25；**修回日期**：2025-11-07
基金项目：国家重点研发计划项目（2022YFB2901700）；国家自然科学基金项目（62372363）
This work was supported by the National Key Research and Development Program of China (2022YFB2901700) and the National Natural Science Foundation of China (62372363).
通信作者：单丹枫（dfshan@xjtu.edu.cn）

中图法分类号 TP393.09

DOI: 10.7544/issn1000-1239.202440909 CSTR: 32373.14.issn1000-1239.202440909

随着互联网的迅速发展,在线观看视频已经成为娱乐的普遍趋势[1]。诸如 YouTube, Netflix, Tiktok 等平台,通过提供高质量的视频内容,吸引了数十亿用户每日在线观看海量的视频内容。根据市场研究报告,2023 年全球视频流媒体市场规模为 5 558.9 亿美元[2]。这种视频观看模式的转变不仅提升了用户的观看体验,也强力驱动着流媒体技术的不断发展。然而,流媒体服务对网络环境的依赖性极高,用户期望在各种网络环境下都能获得流畅且高质量的播放体验。

为了应对网络环境的复杂性和带宽波动,许多视频提供商使用自适应视频流技术来传输视频,例如实时流媒体(HTTP live streaming, HLS)[3] 和基于 HTTP(hypertext transfer protocol)的动态自适应流(dynamic adaptive streaming over HTTP, DASH)[4]。自适应视频流技术是一种根据网络状况和客户端缓冲区占用情况来动态选择视频码率的技术。

在自适应视频流框架中,视频提供商将每个视频分割为多个时长相同的视频块,并为每个块生成若干离散比特率的编码版本。视频播放器可以在任何块边界切换比特率。然后客户端通过码率自适应(adaptive bitrate, ABR)算法使用各种不同的参数(例如预测的可用带宽、缓冲区占用情况等)来动态选择视频块的比特率,并按照顺序请求视频块进行播放。

ABR 算法在播放视频块的同时下载之后的视频块,并将其存储到播放器缓冲区中,播放器以固定的速度播放视频块。但是视频块的下载时间取决于网络条件和视频块的大小。如果缓冲区的视频块在下一个视频块下载完成之前全部播放结束,那么播放器不得不停止播放以等待下一个视频块的到达,从而引起播放卡顿.ABR 算法的目标是为视频块选择合适的码率序列以提供最高的用户体验质量(quality of experience, QoE),包括最大化码率、降低卡顿时间、避免频繁的码率波动等[5-6]。然而这些 QoE 指标中有许多是相互冲突的[7-8],如选择最高可用码率可能导致视频播放卡顿严重,最小化卡顿时间可能导致视频码率低。并且不同用户对这些 QoE 指标的偏好差异很大。有研究表明,各项指标还会影响用户继续观看视频的意愿。对于 90 min 的直播活动,卡顿率增加 1% 会导致用户观看时间降低 3 min 以上[5],频繁的码率切换可能导致用户放弃观看。因此,ABR 算法必须平衡好各项 QoE 指标。

近年来,大量 ABR 算法[1,7,9-18] 被提出,为提升视频观看的 QoE 带来了显著的进步。这些 ABR 算法通常基于应用层信息来进行码率选择。但本文发现,应用层所观测到的网络特征与实际网络特征存在一定偏差,无法准确估计出视频块下载时间,从而影响了用户的视频观看体验。具体而言,现有 ABR 算法在计算视频块的传输时间时通常只考虑了带宽,忽略了往返时延、丢包率等信息的影响,导致下载时间预测误差较大。此外,现有算法大多通过 HTTP 请求和响应测得的带宽等应用层信息。这种信息往往滞后于实际网络状况,尤其在带宽波动剧烈的环境中,网络状态预测的准确性不足会导致频繁的码率切换和播放卡顿,严重影响用户观看体验。

为解决以上问题,本文提出了一种基于传输层信息的视频码率自适应算法 Prophet.与现有算法不同,Prophet 不再从应用层获取带宽,而是从传输层获取准确的网络特征,包括带宽、丢包率、往返时延、探测超时[19-20](即 RACK-TLP[20] 算法定义的 *PTO*(probe timeout)计时器长度)等。基于这些网络特征,Prophet 对视频块的下载过程进行建模,并分析了丢包重传、尾部时延对下载时间的影响。最后,Prophet 根据该模型预测出视频块下载时间,并将其集成到码率选择模块中。

本文基于 dash.js[21](DASH 标准的参考开源实现)实现了 Prophet 算法和其他对比算法,在真实网络环境中测试了 Prophet 在多种场景下的表现。本文还在 20 种不同网络条件下评估了 Prophet 和其他算法,共计进行了 79 h 的视频播放实验。实验结果表明:在所有场景中 Prophet 均优于现有算法,平均 QoE 提高了 0.3%~117.9%。此外,在弱网环境等 5 种场景中,Prophet 均表现优秀,其中弱网环境中平均 QoE 提升 19.6%~46.5%,蜂窝网络中平均 QoE 提升 31.7%~117.9%。

综上所述,本文做出了 4 个贡献:

1)分析和验证了现有 ABR 算法对于下载时间预测的不准确的原因;

2)将传输层实时带宽等信息整合到了视频块下载时间预测模型中,显著提升了预测精度;

3)提出了一种考虑传输层丢包重传和尾部时延影响的 3 阶段视频块下载时间预测模型,克服了应

用层无法感知准确网络特征的局限性；

4)建立了一个基于传输层信息的视频码率自适应框架，结合了下载时间预测模型和现有码率选择策略。

1 相关工作

近年来，ABR 算法迅速发展。现有的 ABR 算法大致可分为 2 类：基于规则的算法和基于学习的算法。

基于规则的算法通常根据观察到的网络带宽和当前视频播放器缓冲区占用情况来决定每个视频块的比特率。具体来说，基于规则的算法可以进一步细分为基于速率的算法[7,9-10]、基于缓冲区的算法[11-12]和基于混合信息的算法[13]。

基于速率的算法，如 FESTIVE[7]，PANDA[9]，CS2P[10]，根据过去几个视频块的下载时间估计网络带宽，然后选择不超过带宽的最高比特率进行视频块传输。然而，由于网络带宽的瞬时变化难以预测，如何准确地估计网络带宽是个巨大的挑战，所以这些算法在带宽剧烈波动的环境中往往表现不佳。

基于缓冲区的算法，如 BBA[11] 和 BOLA[12]，根据客户端缓冲区占用情况来进行比特率选择。这类算法通过保持缓冲区中足够的视频块数量，确保在网络不稳定时仍然能够保持视频继续播放，从而减少卡顿时间。然而，这类算法的问题在于，缓冲区的状态无法准确及时地反映当前的网络状况，其往往需要经过多个视频块的时间之后才能意识到网络的变化。并且在带宽长期波动时，这类算法存在整体 QoE 较低和不稳定问题[22]。

基于混合信息的算法，如 MPC[13]，根据可用带宽、缓冲区占用情况、视频块持续时间等指标的组合来选择码率.MPC 通过可用带宽和缓冲区占用情况等信息来选择码率，并将视频块码率选择问题表述为最大化 QoE 问题。它还提出了一个综合的 QoE 指标，即平均视频质量、平均质量变化、重新缓冲时间和启动延迟的加权组合。

近年来，基于学习的算法迅速发展，如 Pensieve[14]，Comyco[1]，Oboe[15]，Fugu[16]，DRLA[23]，ABRTree[24]。Pensieve[14] 采用深度强化学习技术，通过观察过去决策的结果来学习和做出决策。Comyco[1] 使用模仿学习来学习策略。Fugu[16] 使用神经网络预测视频块下载时间，使用 MPC 作为控制策略。然而，部署困难、性能开销大等缺点制约了这些算法的落地[25]。

实时流媒体的码率自适应与点播场景有些不同，延迟是实时流媒体传输过程中最重要的指标[26-27]。因此 LL-GABR[28] 使用深度强化学习的方法，根据感知视频质量而不是比特率来对 QoE 进行建模，降低了移动设备的能耗。

随着近年来快手、抖音等短视频平台的兴起，短视频领域的码率自适应工作也受到了很多人的关注。与视频点播等传统长视频不同，短视频客户端还会提前加载用户推荐队列中的视频(即用户滑动屏幕后播放的视频)来保证用户切换视频时的 QoE.但是如果用户提前切换视频，预加载的内容未被完全播放，可能会导致巨大的带宽浪费。PDAS[17]，Incendio[18] 等算法通过引入用户在视频的某一时间愿意继续观看的比例来进行更准确的 QoE 预测。

2 动　　机

现有 ABR 算法通常基于应用层信息来进行码率选择。但是应用层所观测到的网络特征与实际网络特征存在显著偏差，从而导致下载时间预测误差较大。这是因为应用层信息(例如 HTTP 下载吞吐量)往往是滞后的、宏观的，并且容易受到应用层协议开销以及传输层复杂机制(如丢包重传、拥塞控制)的掩盖，难以准确捕捉到底层网络的瞬时动态。与此形成对比的是，传输层能够直接提供更实时、更细粒度的网络特征，例如精确的往返时延、实时的丢包率以及当前的拥塞窗口状态等。鉴于此，应用层预测的不准确性主要源于以下几个因素：

1)忽略了往返时延对预测视频块下载时间的影响。现有 ABR 算法通常基于应用层可用带宽来预测视频块的下载时间，但这些算法忽略了往返时延对预测的影响。如图 1 所示，从视频播放器客户端的视角来看，单个视频块的传输过程可描述如下：假设客户端在 T_{request} 时间点发送 HTTP 请求，在 T_{response} 时间点收到了服务器所发送视频块的第 1 个字节，两者之间时间间隔为 1 个网络往返时延(表示为 RTT)，即 $T_{\text{response}} = T_{\text{request}} + RTT$。最终，客户端在 T_{finish} 时间点收到视频块的所有字节。

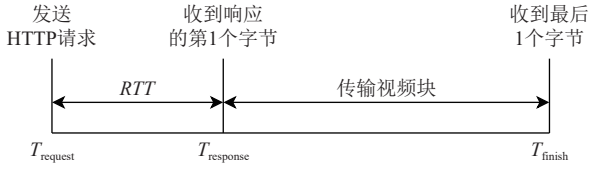


Fig. 1 Illustration of transmission time of video chunk

图 1 视频块传输时间示意图

根据该过程，视频播放器客户端主要通过以下 2 种方法来估计网络带宽：

方法 1。客户端估计带宽的公式为

$$B_1^x = S^x / \left(T_{\text{finish}}^x - T_{\text{response}}^x\right), \tag{1}$$

其中S^x为第x个视频块的大小，B_1^x是该方法所估计的第x个视频块传输时的带宽。基于带宽估计，ABR 算法根据下式来估计下一个视频块的下载时间：

$$T^{x+1} = S^{x+1} / B_1^x, \tag{2}$$

其中T^{x+1}为第$x+1$个视频块的下载时间，S^{x+1}为第$x+1$个视频块的大小。

该方法为 dash.js[21] 默认使用的带宽估计算法，能较为准确地估计出当前的网络带宽，但其对于下载时间的估计并不准确。这是由于该方法在估计下一个视频块的传输时间时未考虑第 1 个字节的传输时间(即$T_{\text{response}} - T_{\text{request}}$)。换而言之，视频块的实际下载时间为$S^{x+1}/B_1^x + RTT$，而非式 (2)。因此，利用该方法仅在视频块下载时间(即T^{x+1})远大于 RTT 时才能准确估计下载时间。而在 RTT 与视频块传输时间相当时，该方法会造成下载时间预测偏小。

图 2 所示为在实际网络中视频传输的测试结果，在带宽为 2 Mbps，丢包率为 0%，RTT 为 50~400 ms 的网络环境中，MPC 算法的预测误差随着 RTT 的增大而逐渐加剧。

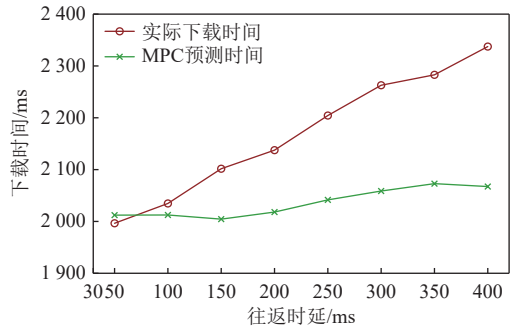


Fig. 2 download time predicted by MPC and actual download time

图 2 MPC 预测的下载时间和实际下载时间

方法 2。与方法 1 不同，该方法在估计带宽时考虑了第 1 个字节的传输时间，其估计带宽的公式为

$$\begin{aligned} B_2^x &= S^x / \left(T_{\text{finish}}^x - T_{\text{request}}^x\right) = \\ &S^x / \left(T_{\text{finish}}^x - T_{\text{request}}^x + RTT\right) = \\ &S^x / \left(S^x / B_1^x + RTT\right), \end{aligned} \tag{3}$$

其中B_2^x是使用该方法计算出的第x个视频块下载时的带宽，T表示客户端请求视频块的时间。根据上述带宽估计，使用下式来估计下一个视频块的下载时间：

$$\begin{aligned} T^{x+1} &= S^{x+1} / B_2^x = \\ &S^{x+1} / B_1^x + RTT \times S^{x+1} / S^x \text{。} \end{aligned} \tag{4}$$

该方法虽然能准确估计出应用层吞吐量。然而，该方法仅在视频块大小相同时才能准确估计下载时间。在视频块大小不同时，会产生 1 个与 RTT 成正比的误差。具体地，实际视频块下载时间应为

$$T^{x+1} = S^{x+1} / B_1^x + RTT \text{。} \tag{5}$$

不难看出，只有在$S^{x+1} = S^x$时，式 (4) 才和式 (5) 等价。然而，视频块大小与码率直接相关，码率越高，视频块占用空间越大。此外，当码率相同时，不同视频块大小也不一致[29]。因此在网络出现突发拥塞或码率切换较频繁时，采用这种方法计算的预测误差将比方法 1 更大。

综上所述，视频块下载时间会受到往返时延的影响，目前根据应用层带宽观测的方法无法准确地估计出视频块下载时间。无论客户端采用哪种方式来计算带宽，都存在 1 个与 RTT 成正比的误差。对于 RTT 较小的情况，这种误差可以忽略不计；但在 RTT 较大，尤其是弱网环境中，RTT 可能占视频块下载时间的 20% 以上，此时现有的算法对于视频块下载时间的预测会显得非常不准确。

2)忽略了丢包对计算应用层带宽的影响。现有 ABR 算法大多基于应用层可用带宽来选择码率，而客户端基于视频块的下载时间来计算可用带宽。这些算法忽略了丢包通过影响下载时间来间接影响带宽计算。尾部时延是影响视频块下载时间的重要因素，如图 3 所示。但尾部时延的长度取决于丢包率和丢包位置，且丢包位置具有随机性，因此视频块传输时间也随着丢包位置的变化而变化。客户端使用下载时间的瞬时值来计算带宽，带宽计算值随着丢包率和丢包位置的改变而波动。在丢包率相同的情况下，若丢包发生在视频块传输过程的末尾，其下载时间可能比发生在开头长 20% 以上，带宽计算值也随之缩小。

如图 4 所示，在丢包率分别为 0% 和 10% 的场景下测量了 dash.js[21] 计算的带宽，在丢包率为 0% 时，dash.js 测量的带宽波动较小；在丢包率为 10% 时，带宽波动明显增大。实验结果表明，仅丢包位置的随机变化就能让客户端测量的带宽出现明显的误差。

综上所述，应用层带宽计算受到了传输层信息(即丢包)的影响。目前在应用层无法准确计算带宽，只能通过加权平均等方法来减少误差。

3)实时响应能力差。由于在应用层不能保证带

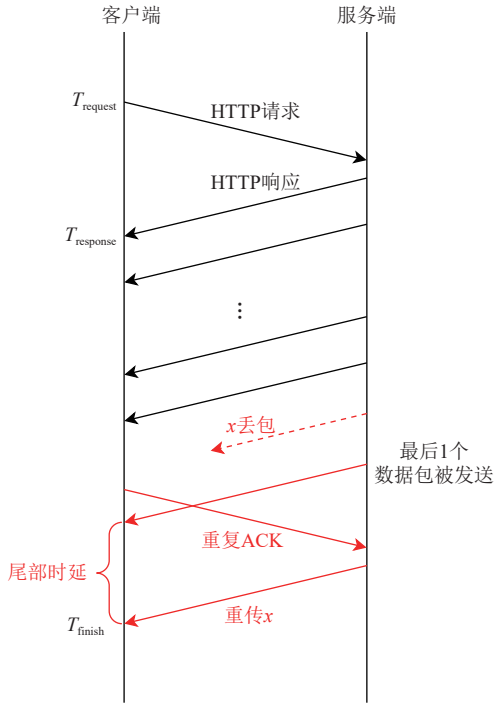


Fig. 3 Illustration of tail delay

图 3 尾部时延示意图

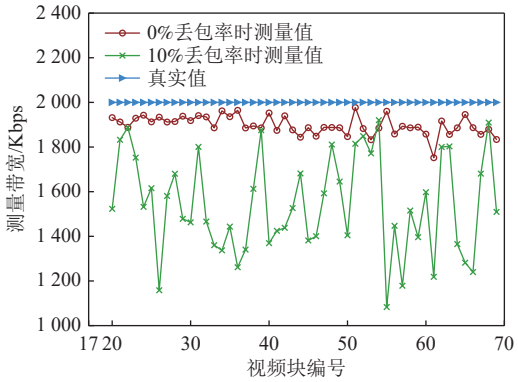


Fig. 4 Bandwidth measured by dash.js at different packet loss rates

图 4 不同丢包率时 dash.js 测量的带宽

宽的准确计算，所以现有客户端通常采用加权平均等方式来计算平均带宽。然而，面对突发的拥塞或丢包，应用层估计的平均带宽难以准确、及时地反映这些变化，往往在传输层感知到拥塞后经过较长时间才能发现问题[30]。于是，客户端无法在发送下一个视频块请求时及时调整比特率，最终导致卡顿。例如，在视频播放过程中，当带宽突然从 3 Mbps 降为 1.5 Mbps 时，MPC 算法在应用层估计的带宽需要经过下载 5 个视频块的时间才趋于稳定，如图 5 所示。

综上所述，现有的 ABR 算法对于下载时间的估计不够准确，根本原因在于：应用层所观测到的网络特征与实际网络特征存在一定偏差，只通过可用带

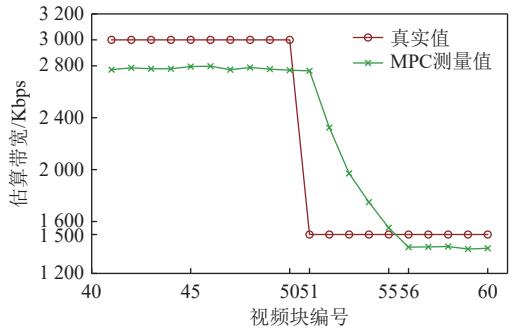


Fig. 5 Estimated bandwidths of MPC change with variation of network environment

图 5 网络环境变化后 MPC 估算带宽的变化

宽不能准确地评估网络状况。于是，本文提出了一种基于传输层信息的 ABR 算法。该算法通过读取传输层的网络信息，更准确地反映实时网络特征。此外，本文基于传输层信息对于视频块下载过程进行建模，分析了丢包重传、尾部时延等因素对下载时间的影响，使客户端能够更准确地选择码率。

3 Prophet 设计

本节提出了一种基于传输层信息的视频码率自适应算法 Prophet。该算法的核心思路是基于传输层信息对视频块的下载过程进行精细建模，从而更准确地评估网络特征，并据此指导视频码率的选择，以提升 QoE。

在模型设计中，Prophet 利用传输层的多维网络信息，包括网络带宽、丢包率、往返时延以及探测超时，这些信息均通过服务端在传输层实时测量获得。本文将上述信息视为在视频块下载过程中保持稳定的输入变量，且相互之间独立。尽管在实际网络中，这些参数可能存在一定的耦合关系（例如丢包可能影响往返时延测量值，或高 *RTT* 可能延长丢包重传时延），但本文的模型建立并未依赖这种耦合效应，不会影响后续公式推导的准确性。

具体建模过程如下：假设完整视频由 M 个连续的视频块组成，每个视频块长度为 L（单位为 s），编号为 $\{1, 2, \cdots, N\}$。每个视频块支持多个码率版本，令 R_k 表示第 k 个视频块的比特率，$S_k(R_k)$ 表示该码率下的视频块的大小，$T_k(R_k)$ 表示下载该视频块所需的时间。

如图 6 所示，视频块的传输过程可划分为 3 个阶段：

启动阶段。客户端发起请求至接收到服务器返

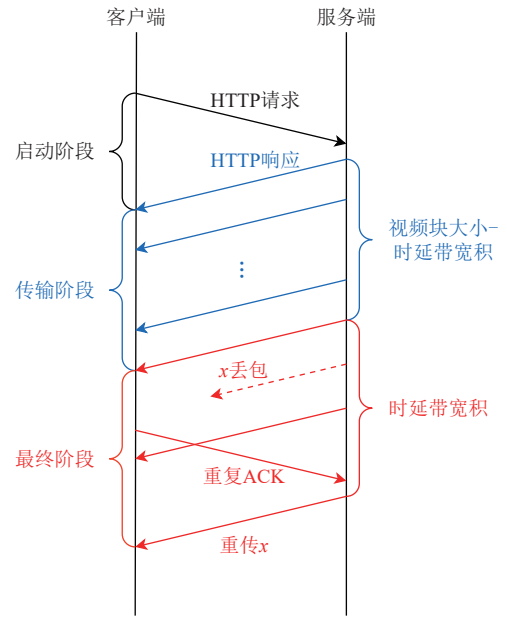


Fig. 6 Three stages of video chunk transmission process

图 6 视频块传输过程的 3 个阶段

回的首个字节，主要由往返时延决定，记作t_1。

传输阶段。视频块的主体数据传输过程，传输除最后 1 个 *BDP*(bandwidth-delay product)大小的数据以外的部分，传输时间记作t_2。

最终阶段。视频块中最后 1 个 *BDP* 大小的数据的传输过程，受尾部丢包重传等影响，传输时间记作t_3。

综上，第k个视频块在码率为R_k时的总下载时间为

$$T_k(R_k)=t_1+t_2+t_3\ 。\tag{6}$$

随后，本文将基于丢包率和传输层重传机制(包括快速重传[31]、早期重传[32]、FACK[33]、SACK[34]、RACK-TLP[20]等)进一步分析各阶段传输时间的计算方式，最终形成完整的视频块下载时间预测模型。

3.1 启动阶段

该阶段从客户端发出请求开始，到接收到响应的第 1 个字节结束。该阶段的传输时间主要由往返时延(用 *RTT* 表示)决定。该阶段下载时间计算公式为

$$t_1=RTT\ 。\tag{7}$$

3.2 传输阶段

本文将视频块数据的传输过程划分为传输阶段和最终阶段，原因在于：传输阶段丢包不会产生尾部时延，而最终阶段丢包一定会产生尾部时延。

具体来说，传输阶段的数据包如果发生丢包，触发快速重传需要 1 个 *RTT* 的时间。即使最终阶段未发生丢包，其传输时间至少为 1 个 *RTT* 的时间，如果发生丢包则更长。所以传输阶段的数据包触发快速重传的时间一定早于最终阶段的最后 1 个数据包的发送时间，如图 7 所示。因此传输阶段的数据包的第 1 次丢包不会导致尾部时延，只会导致视频块总传输时间延长发送 1 个数据包的时间。对于传输阶段的最后 1 个 *BDP* 中发送的多次丢包，由于其第 1 次重传包发送在最终阶段第 1 个包之后，于是可以将其划分入最终阶段，等价为最终阶段多发一定数量的包，数量为传输阶段的最后 1 个 *BDP* 中发生 1 次丢包的包数。而最终阶段的数据包会在 1 个 *RTT* 的时间内传输完成，一旦发生丢包，重传需要至少 1 个 *RTT* 的时间，即只要发生了丢包，该数据包进行重传的时间一定在发送最后 1 个数据包的时间之后，如图 8 所示。

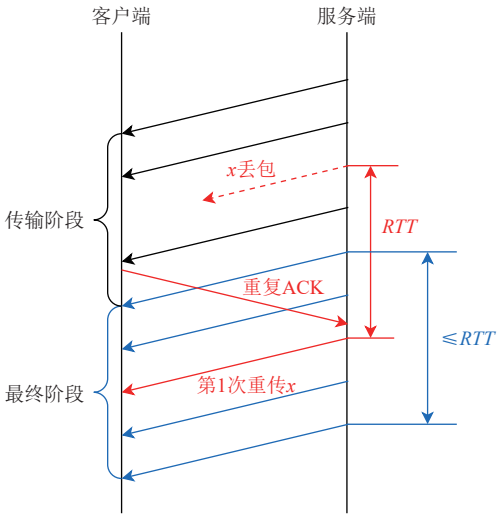


Fig. 7 Illustration of packet loss and retransmission in the transmission phase

图 7 传输阶段的丢包重传过程示意图

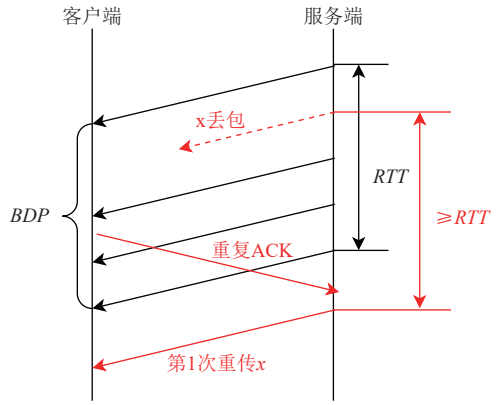


Fig. 8 Packet loss and retransmission in the final phase

图 8 最终阶段的丢包重传过程

综上所述，该阶段下载时间为

$$t_2=(S-C\times RTT)/C/(1-p)+RTT, \tag{8}$$

其中S为视频块大小，C为从传输层读取的带宽，p为从传输层读取的丢包率。

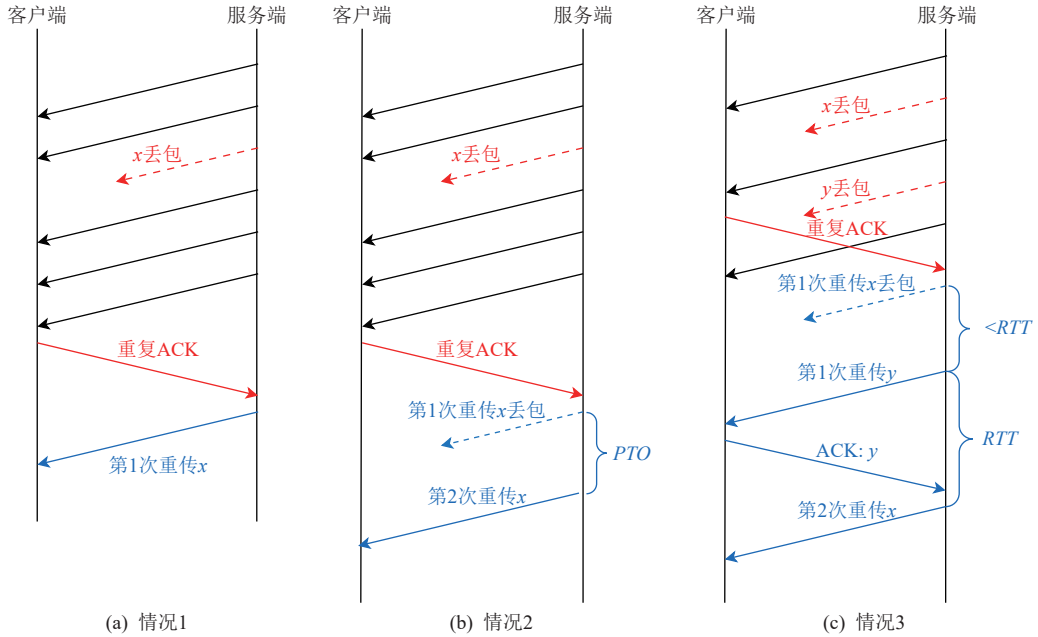
3.3 最终阶段

最终阶段中传输的是视频块最后1个*BDP*大小的数据。假设该阶段一共有N个数据包，将其编号为{1, 2, …, N}，包括了最后1个*BDP*大小的数据和传输阶段由于丢包而进入最终阶段的包。N的计算公式为

$$N=C\times RTT/S_{\text{packet}}+p\times C\times RTT/S_{\text{packet}}, \tag{9}$$

其中S_{packet}为1个数据包的大小。

由于最终阶段的数据包进行重传的时间一定在发送最后1个数据包的时间之后，所以该阶段完成时间只与最后1个丢包的数据包的重传时间有关。本文根据每个包的丢包次数将该部分的传输过程分为3种情况讨论，如图9所示。情况1为所有数据包最多连续丢1次，情况2和情况3为所有数据包最多连续丢2次。



(a) 情况1 (b) 情况2 (c) 情况3

Fig. 9 Three scenarios in the final stage

图9 最终阶段的3种情况

3.3.1 情况1：最多丢1次

如图9(a)所示，情况1为每个数据包最多丢1次的场景：假设最后1个丢的包是x号包，即1号包到$x-1$号包最多丢1次，x号包丢1次，$x+1$号包到N号包没有丢包。已知丢包概率为p，该情况发生概率为

$$P_{1,x}=\left(1-p^2\right)^{x-1}\times p(1-p)\times(1-p)^{N-x}=p(1-p)^{N-x+1}\left(1-p^2\right)^{x-1}, \tag{10}$$

其中$p_{1,x}$表示x号包发生情况1的概率。此时x号包第1次传输后至少还有$N-x$个包未发送，所以一定能触发快速重传.TCP(transmission control protocol)和QUIC(quick UDP internet connections)都采用了时间阈值和ACK结合的丢包检测和快速重传机制，以减少对重复ACK数量的依赖。基于此，传输时间为

$$T_{1,x}=\min\{RTT+kT_{\text{packet}}|k=1,2,3\text{且}\ RTT+kT_{\text{packet}}>T_{\text{thre}}\}+(x+1)T_{\text{packet}}, \tag{11}$$

其中T_{packet}是发送1个数据包的时间，T_{thre}是RFC8985或RFC9002中定义的时间阈值。

其中N号包比较特殊，由于其是最后1个数据包，如果丢包后没有其他数据包再丢包，其不会触发快速重传，而是基于TLP算法在*PTO*计时器到期后重传，所以N号包的尾部时延会比其他数据包更高。建模时应将N号包丢包的情况单独处理。此时1号包到$N-1$号包最多只丢1次，N号包丢1次，该情况发生概率为

$$P_{1,N}=\left(1-p^2\right)^{N-1}\times p(1-p)=p(1-p)\left(1-p^2\right)^{N-1}。 \tag{12}$$

与之前的情况不同，只有重传包才会发送在N号包之后，所以如果前$N-1$个包都未丢包，则N号包只能等待*PTO*计时器到期后才能进行重传；如果前面有丢包，则N号包可以通过重复ACK触发快速重传。RACK-TLP算法引入了时间阈值来判断丢包，只要超过时间阈值，即使只有1个重复ACK也能触发

快速重传，但是考虑到实际传输中连续丢包的概率很低，所以可以近似认为每一次丢包后收到重复ACK的时间超过了时间阈值，即只要收到1个重复ACK，就判断之前发的包全部丢包并进行重传。为简化计算，可以将x号包触发快速重传的时间估计为其上界（即2个包各自进行第1次重传的间隔的最大值），即认为x号包通过$x-1$号包的2次丢包的ACK触发重传。因此传输时间为

$$T_{1,N}=(1-p)^{N-1}\left((N+1)T_{\text{packet}}+PTO\right)+\left(1-(1-p)^{N-1}\right)\left((N+1)T_{\text{packet}}+2RTT\right),\quad(13)$$

其中PTO表示传输层读取的探测超时。

3.3.2 情况2：最多丢2次且无法快速重传

情况2和情况3均为所有数据包最多连续丢2次，假设最后1个连续丢包2次的包为x号包。x号包第1次丢包后是否有其他数据包再次丢包会决定x号包触发第2次重传的方式为快速重传还是TLP重传，可以以此来区分情况2和情况3。如图9(b)所示，情况2为x号包第1次丢包后没有其他数据包丢包，即1号包到$x-1$号包最多丢1次，x号包连续丢2次，$x+1$号包到N号包没有丢包。该情况发生概率为

$$P_{2,x}=\left(1-p^2\right)^{x-1}\times p^2(1-p)\times(1-p)^{N-x}=p^2(1-p)^{N-x+1}\left(1-p^2\right)^{x-1}。\quad(14)$$

此时x号包第1次丢包与情况1相同，仍然是触发快速重传。x号包的第1次重传包就是最后1个发送的数据包，没有重复ACK来触发快速重传，所以需要等待PTO计时器结束才能重传。传输时间为

$$T_{2,x}=T_{1,x}+PTO+T_{\text{packet}}。\quad(15)$$

3.3.3 情况3：最多丢2次且能快速重传

如图9(c)所示，情况3为x号包连续丢包2次，且x号包的第1次丢包和第2次丢包之间还有其他数据包丢包，即1号包到$x-1$号包最多丢2次，x号包连续丢2次，$x+1$号包到N号包最多丢1次，且需要排除掉情况2.该情况发生概率为

$$P_{3,x}=\left(1-p^3\right)^{x-1}\times p^2(1-p)\left(1-p^2\right)^{N-x}-\left(1-p^2\right)^{x-1}\times p^2(1-p)\times(1-p)^{N-x}=p^2(1-p)\left(1-p^2\right)^{N-x}\left(1-p^3\right)^{x-1}-p^2(1-p)^{N-x+1}\left(1-p^2\right)^{x-1}。\quad(16)$$

与情况1的N号包类似，此时x号包第2次丢包后通过重复ACK触发快速重传。同样将x号包触发快速重传的时间估计为其上界。因此传输时间为

$$T_{3,x}=T_{2,x}+2RTT+T_{\text{packet}}。\quad(17)$$

3.3.4 最终阶段传输总时长

为了计算最终阶段的平均传输时间，3.3.1~3.3.3节分别枚举每个数据包作为最后1个被重传数据包的3种可能情形，计算其对应的概率（记为$P_{1,i}$，$P_{2,i}$，$P_{3,i}$）与传输时间（记为$T_{1,i}$，$T_{2,i}$，$T_{3,i}$），并对每个数据包作为最后1个被重传数据包的期望传输时间求和，从而获得最终阶段的期望传输时间为

$$t_3=\sum_{i=1}^{N}\left(P_{1,i}T_{1,i}+P_{2,i}T_{2,i}+P_{3,i}T_{3,i}\right)。\quad(18)$$

3.4 视频块传输总时长

综上所述，根据式(6)~(8)，(18)，码率为R_k的第k个视频块下载时间$T_k(R_k)$计算公式为

$$T_k(R_k)=2RTT+(S-C\times RTT)/C/(1-p)+\sum_{i=1}^{N}\left(P_{1,i}T_{1,i}+P_{2,i}T_{2,i}+P_{3,i}T_{3,i}\right),\quad(19)$$

其中RTT表示从传输层读取的往返时延，S表示视频块的大小，C表示从传输层读取的带宽，p表示从传输层读取的丢包率。$P_{1,i}$，$P_{2,i}$，$P_{3,i}$分别表示最终阶段第i个数据包发生情况1、情况2、情况3的概率，可根据式(10)(12)(14)(16)计算。$T_{1,i}$，$T_{2,i}$，$T_{3,i}$分别表示最终阶段第i个数据包发生情况1、情况2、情况3时的最终阶段传输时间，可根据式(11)(13)(15)(17)计算。

3.5 码率选择

在估计出视频块下载时间之后，需要根据该时间选择恰当的码率。在码率选择算法方面，本文延续了业界主流的策略，直接使用了被CS2P[10]，Fugu[16]等广泛使用的MPC[13]作为控制策略，该策略通常以吞吐量预测、当前比特率和缓冲区占用情况作为输入，并通过解决一个QoE最大化问题来动态确定下一个视频块的码率。

Prophet算法在此基础上进行了改进，使用本文提出的基于传输层信息的视频块下载时间预测模型来取代MPC中原有的通过吞吐量预测和当前比特率来计算下载时间的过程。Prophet提供的高精度预测值作为MPC的更可靠输入，从根本上避免了传统MPC因吞吐量预测不准确而导致的性能不稳定问题，例如过度激进的码率选择引发卡顿，或过度保守的码率选择浪费带宽。

Prophet算法通过提供准确、实时的传输层参数和精准的下载时间预测，显著增强了MPC优化QoE的能力。因此，传输层特征并非独立于MPC的QoE目标，而是作为MPC的“眼睛”，为其提供更强的网络观察能力，做出更智能、更高效的码率决策。

4 实　　验

本节设计了一个基于传输层信息的视频码率自适应框架，并在真实网络环境中搭建了一个视频播放系统，将 Prophet 和现有的算法进行比较。

4.1 实　现

本节详细阐述了 Prophet 算法的整体系统框架及传输层信息的获取机制，如图 10 所示。整个框架在逻辑上清晰地划分为客户端和服务端两大部分。

服务端的核心功能在于实时获取传输层网络参数。Prophet 作为一个轻量级的模块，不涉及传输层协议的重新实现或额外状态的维护，而是通过与传输层协议栈交互，直接读取和整理其内部已维护并暴露的、用于自身拥塞控制和丢包恢复的实时状态量和统计数据。这些从传输层获取到的带宽等网络参数，随后会被高效且及时地反馈给客户端。

客户端运行于应用层，主要由以下功能模块协作完成码率自适应任务。网络信息收集模块负责接收并标准化来自服务端的实时网络信息。随后，下载时间预测模块根据这些精准的传输层网络参数，运用本文提出的下载时间预测模型，精确估算出下一个视频块的下载时间。最后，码率决策模块基于下载时间预测结果和播放缓存模块的缓存占用情况，选择能使 QoE 最大化的视频码率。

Prophet 旨在高效地利用传输层协议自身已计算和维护的关键参数。它直接从传输层协议栈（如 QUIC 的 XQUIC[35] 实现）中读取瞬时交付速率（作为带宽）、*RTT*、*PTO* 等数据。针对丢包率的获取，如果传输层协议本身未直接提供该数据，Prophet 则会根据协议栈暴露的原始丢包和确认包数量统计进行计算。具体而言，服务端会记录每个数据包发送时累计丢包数量和累计已收到 ACK 确认的包的数量同时，也会记录收到特定 ACK 时对应的当前累计丢包数量和累计已确认包数量。瞬时丢包率由以下公式计算得出：

$$p = (L_1 - L_2)/(A_1 - A_2), \tag{20}$$

其中 L_1 和 A_2 分别为当前时刻的累计丢包数量和累计确认包数量，L_1 和 A_2 为上一测量点的累计丢包数量和累计确认包数量。为避免丢包率的剧烈波动影响计算准确性，可以对其样本采用指数移动平均值进行平滑处理，计算公式如下：

$$p_{\mathrm{sm}} = \frac{7}{8} \times p_{\mathrm{sm}} + \frac{1}{8} \times p, \tag{21}$$

其中 p_{sm} 表示平滑丢包率，p 表示瞬时丢包率.Prophet 仅对这些由传输层实时计算和维护的参数进行利用，无需引入额外的计算复杂性或状态存储。

为实现上述传输层信息的实时采集，本文在服务端采用了 QUIC 协议，并基于开源的 XQUIC 实现。QUIC 协议的拥塞控制和丢包检测机制完全运行在用户态，开发者能够以极低的访问成本精确采集网络状态。这种用户态设计极大地方便了 Prophet 直接访问并读取传输层内部的关键状态变量，从而保证了参数的实时性和准确性。相比之下，传统 TCP 协议运行在内核态，其拥塞控制逻辑与统计信息由操作系统内核维护。尽管现代 TCP 同样支持精确的往返时延测量和丢包检测机制，但要实时获取所需参数的难度相对更高，通常需要通过系统调用来间接获取。

4.2 实验设置

实验环境基于真实网络搭建，其中使用了 1 台 Windows 主机作为视频播放器客户端，1 台云服务器作为服务端。在服务端，使用 QUIC 协议作为传输层协议。QUIC 协议的实现方案使用 XQUIC，结合 Tengine[36] 作为与 XQUIC 配套的 HTTP 服务器，客户端和服务器之间的视频块传输使用 HTTP/3 协议。

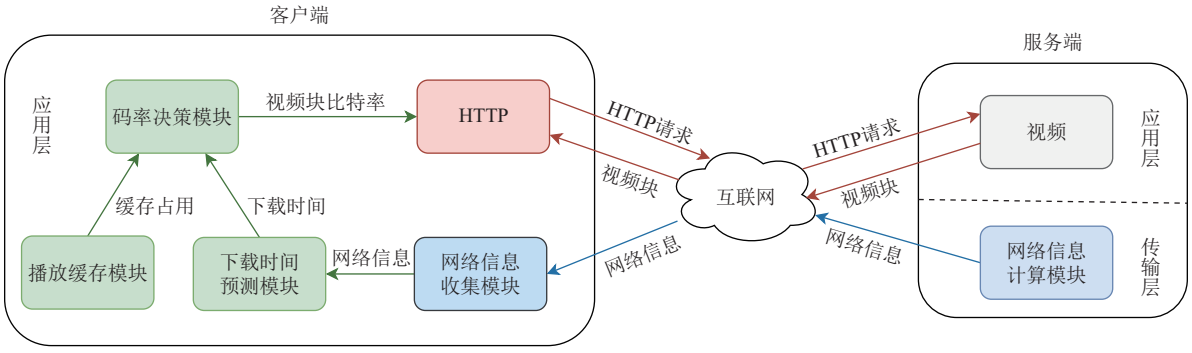


Fig. 10　Overall framework for video bitrate adaption based on transport layer multi-dimensional information

图 10　基于传输层多维信息的视频码率自适应总体框架

此外, 为解耦视频块的传输与网络信息的传输, 所以服务端通过 QUIC 协议来传输视频块, 通过 TCP 连接将网络信息传输给客户端。在客户端, 基于 dash.js 框架来构建视频播放器客户端, 并使用 Firefox 播放视频。在服务端使用 Linux 提供的流量控制工具 tc 来控制网络环境。

视频参数: 视频使用的是 dash.js 参考客户端中的“EnvivoDash3”视频[37]。该视频使用 H.264/MPEG-4 编码器编码, 提供 6 个比特率版本: 300 Kbps, 750 Kbps, 1 200 Kbps, 1850 Kbps, 2 850 Kbps, 4 300 Kbps。该视频共包含 97 个时长为 2 s 的视频块, 总时长为 194 s。

QoE 指标: 用户对视频流 QoE 的喜好是存在差异的, 有些用户更重视视频质量, 也有些用户更关注视频播放流畅。仅最大化带宽利用率并不能保证最佳的 QoE[38], 因为视频比特率过高很可能导致视频卡顿。因此, ABR 算法需要在不同用户之间取得平衡。本文使用的是得到广泛应用的 Qo E 指标, 该指标由 MPC 提出。其计算公式如下:

$$QoE_1^N=\sum_{n=1}^{N}q(R_n)-\lambda\sum_{n=1}^{N-1}|q(R_{n+1})-q(R_n)|-\mu\sum_{n=1}^{N}(T_n(R_n)-B_n),\tag{22}$$

其中N表示视频块总数, R_n表示第n个视频块的比特率, $q(R_n)$表示该比特率对应的视频质量, $|q(R_n)-q(R_{n+1})|$表示对视频码率变化的惩罚, $T_n(R_n)$表示下载码率为R_n的第n个视频块的下载时间, B_n表示下载第n个视频块时的已缓存时间。λ, μ表示对于码率变化和播放卡顿的惩罚系数。

自适应算法: 本文将 Prophet 与以下算法比较。

1) MPC[13]: 使用大小为 5 的前瞻窗口, 通过缓存占用和带宽预测来选择比特率。

2) BOLA[12]: 将 ABR 问题转化为效用最大化问题, 利用 Lyapunov 函数求解。这是一种基于缓冲区的方法。本文使用 dash.js 提供的 BOLA 版本[37]。

3) Pensieve[14]: 使用深度学习来选择码率。本文使用的是作者提供的实现[39]。

4.3 模型验证

本节将验证下载时间预测模型的准确性。实验环境设置为带宽 2 Mbps, 丢包率 10%, *RTT* 为 200 ms, 针对模型中视频传输过程的 3 个阶段, 逐一验证预测的下载时间的误差。

如表 1 所示, Prophet 对整体视频块下载时间的预测误差低于 8%, 基本实现了对下载时间的准确预测。其中, 整体传输过程的预测误差低于各个独立阶段的误差, 这主要因为 Prophet 在计算传输时间时实际计算的是数学期望。具体来说, 不同阶段的预测误差可能互相抵消, 例如某些阶段的预测值略高于实际值, 而其他阶段则偏低。因此, 正负误差的平衡使得整体传输过程的误差在统计上低于单个阶段的误差。

Table 1 Prediction Errors of Video Chunk Download Time by Prophet at Every Stages

表 1 Prophet 每个阶段视频块下载时间的预测误差 %

阶段	误差
启动阶段	9.81
传输阶段	8.62
最终阶段	13.80
整体传输过程	7.69

然而, 各阶段预测误差仍大于 5%, 这不仅与数学期望的计算方式有关, 还与网络环境的不稳定性有关。视频块下载过程通常持续 1~2 s(假设视频块时长 2 s), 网络环境在下载过程中无法保持稳定, 而 Prophet 在预测视频块的下载时间时无法预测到后续的网络波动。

另外, 最终阶段的误差明显高于其他阶段, 原因在于最终阶段受丢包位置的随机性影响最大。启动阶段和传输阶段的传输时间几乎不受丢包位置的影响。而最终阶段的传输时间可分为数据包首次传输时间和尾部时延 3 部分, 数量级均为 1 个 *RTT*。如果发生概率极低的极端情况, 例如最终阶段某数据包连续多次丢包时, 对于尾部时延部分的计算误差可能超过 100%, 因此最终阶段的误差大于其他阶段的误差。

4.4 场景验证

本节将通过 3 个场景来对比 Prophet 和现有算法, 展示 Prophet 的实际应用效果。

4.4.1 往返时延对下载时间的影响测试

本实验设置了一个场景, 网络参数为带宽 2 Mbps, 丢包率为 0.1%, *RTT* 为 50~400 ms.在该场景中, 分别使用 Prophet 和 MPC 下载比特率为 1 200 Kbps 的测试视频第 50 号视频块, 以比较双方预测的下载时间的误差。

随着 *RTT* 的增大, Prophet 预测的下载时间和实际下载时间基本保持同步增长, 预测误差在 1% 以内, 如图 11 所示。

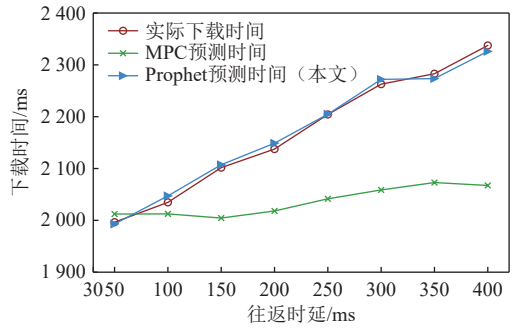


Fig. 11 Download time predicted by different algorithms with different *RTT*s

图 11 不同 *RTT* 不同算法对下载时间的预测

这表明在 *RTT* 较大时，Prophet 算法仍能较为精确地预测下载时间。而 MPC 预测的下载时间受 *RTT* 的影响较小，这与第 2 节的分析一致：在 *RTT* 较小时，由于下载时间远大于 *RTT*，MPC 仍能保持较为准确的预测；然而，随着 *RTT* 的增大，实际下载时间迅速增大，而 MPC 的预测值几乎保持不变，导致预测误差逐渐加大。这种现象的主要原因在于 MPC 忽视了视频块传输的启动阶段。启动阶段时链路并没有达到满载状态，该阶段传输时间主要取决于 *RTT* 而不是带宽。因此在 *RTT* 增大时，实际下载时间逐渐增大，而 MPC 几乎保持不变。

综上所述，随着 *RTT* 的增大，MPC 的下载时间预测误差增大，Prophet 的预测误差基本不变，更适用于高 *RTT* 场景。

4.4.2 丢包对带宽测量的影响测试

本实验设置了丢包率分别为 0% 和 10% 的场景，分别记录了 dash.js 和 Prophet 在选择连续的 50 个视频块的码率时的带宽测量值，以比较丢包对带宽测量的影响。其中 dash.js 在应用层测量带宽，Prophet 在传输层测量带宽。如图 12 所示，在丢包率为 0% 时，2 个算法的带宽测量值波动都很小。

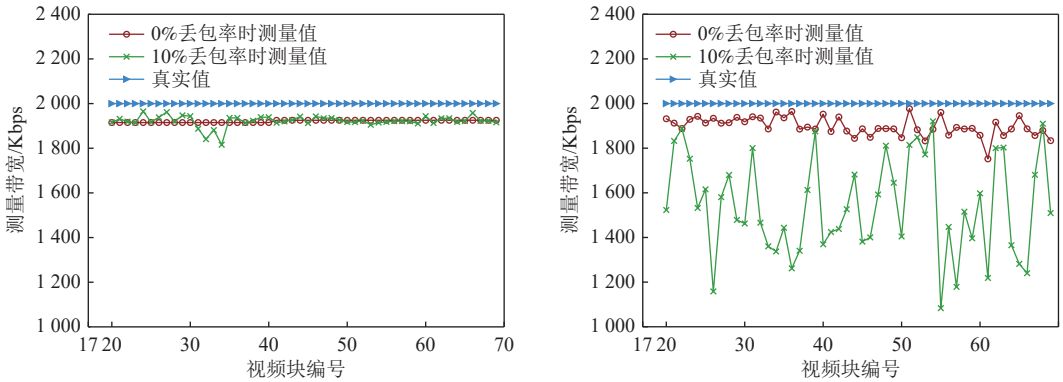
dash.js 在应用层通过视频块的下载时间测量带宽，而应用层计算的时间与传输层计算的时间存在一定延迟。具体而言，应用层发出请求一段时间后传输层才会发出请求，传输层收到视频块后也需要花费一定时间传输给应用层，这些时间都会被应用层统计为视频块的下载时间。因此，在丢包率为 0% 时，dash.js 的测量结果仍会受到这些因素的影响。因此存在微小波动。而 Prophet 在传输层使用 BBR[40] 的方法来计算带宽，测量结果几乎没有波动。

随着丢包率的增大，Prophet 仍然保持极低的波动，而 dash.js 测量的带宽波动幅度显著增大。其主要原因正如第 2 节所说，丢包位置的随机性导致视频块下载时间长度随机波动。而 dash.js 基于下载时间的瞬时值计算带宽，因此间接受到了尾部时延的影响，造成带宽测量的巨大波动.dash.js 还存在几次明显的带宽下降，抓包结果显示，这是由于 3.3 节中提及的某一个数据包连续 2 次丢包导致尾部时延巨幅增大，从而导致测量的带宽显著偏低。而 Prophet 在传输层测量带宽，不受下载时间影响，因此带宽测量值波动极小。综上所述，丢包率的增大对 Prophet 测量带宽的影响很小，而会对 dash.js 造成显著影响。

4.4.3 网络突发拥塞响应速度测试

本实验设定了一个带宽骤降的场景来模拟网络拥塞，在视频播放器收到第 50 号视频块时（即开始选择第 51 号视频块码率时）将服务器的带宽从 3 Mbps 突然降为 1.5 Mbps。通过测量 Prophet 算法和 MPC 在每个视频块进行码率决策时计算的带宽，来比较 MPC 在应用层估计带宽和 Prophet 在传输层估计带宽对于网络突发拥塞的反应速度。

如图 13 所示，在网络发生拥塞后，Prophet 算法



(a) Prophet（本文）测量的带宽 (b) dash.js测量的带宽

Fig. 12 Impact of packet loss rate on bandwidth measurement

图 12 丢包率对带宽测量的影响

在开始下载第 52 号视频块时已检测到网络拥塞并计算出稳定的带宽。而 MPC 直到开始下载 56 号视频块时带宽预测值才趋于稳定。

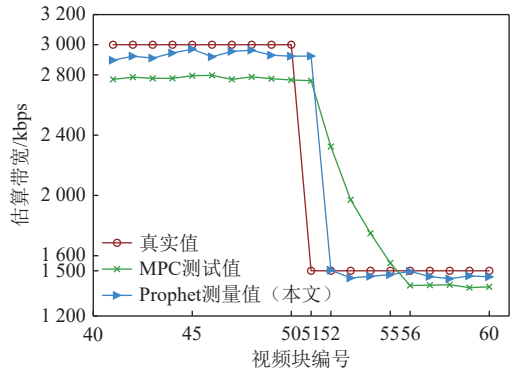


Fig. 13 Changes in bandwidth predicted by different algorithms after bandwidth reduction

图 13 带宽降低后不同算法预测带宽的变化

尽管 MPC 在此期间逐步降低了带宽预测值，但仍明显高于实际带宽，容易引起播放卡顿。该现象的主要原因在于 MPC 在应用层计算带宽，其得出的是视频块下载期间的平均带宽；而 Prophet 直接在传输层计算瞬时带宽，因此对于网络拥塞的响应速度会更加迅速、准确。

4.5 性能测试

本节在真实网络中将 Prophet 和现有算法进行了对比评估，涵盖了 20 种不同的场景，总计播放了 79 h 的视频。参考环境为带宽 2 Mbps、丢包率 10%、*RTT* 为 150 ms，本实验测量了各算法的下载时间预测误差、归一化 QoE 和 QoE 各具体指标等分别随丢包率、*RTT*、带宽变化的趋势。

4.5.1 下载时间预测误差测试

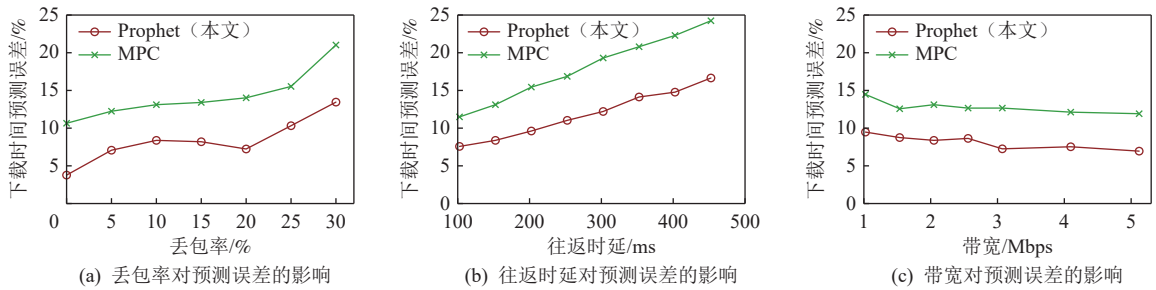
图 14 展示了 Prophet 与其他算法在不同网络条件下对下载时间的预测情况。具体来看，在任何场景下，Prophet 的预测误差均明显低于 MPC，且在大部分场景都能保持在 10% 以下。这一结果表明，Prophet 在应对复杂网络环境时能保持优秀的预测准确性和鲁棒性。

值得说明的是，由于 MPC 忽略了视频块传输的启动阶段，导致其预测的下载时间往往低于实际下载时间，且计算误差较大。这种低估会导致 MPC 更倾向于采用激进的比特率策略，选择高于实际网络条件的码率，从而导致频繁的码率切换和播放卡顿。这种现象在网络环境较差（例如高丢包或者高延迟的环境）时更加明显，其计算误差已经超过了 20%。相对而言，Prophet 在高延迟和高丢包的场景下误差增持较为平缓。这种现象归因于 Prophet 计算的是下载时间的数学期望，使其能够在较差的网络环境中仍提高较为准确的预测。为了进一步提升预测的可靠性并减少导致的播放卡顿，Prophet 对最终阶段的传输时间进行了适度的高估，如 3.3 节所示。通过这种方法，Prophet 在网络波动的环境下，仍能保持视频流的流畅播放，进而提升用户的观看体验。

4.5.2 QoE 测试

图 15 展示了不同算法在不同场景下的归一化 QoE，所有算法的 QoE 均相对于 MPC 的表现进行归一化。实验结果表明，所有网络条件下，Prophet 均优于现有的 ABR 算法，在不同场景下平均 QoE 提高了 0.3%~17.9%。

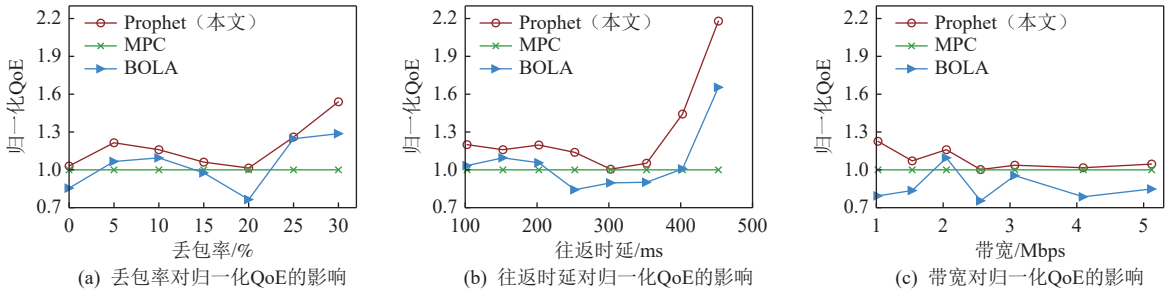
值得一提的是，随着丢包率或 *RTT* 的提高，Prophet 相对于 MPC 的归一化 QoE 呈现一定的非线性变化。主要原因在于视频可选择的码率并不连续，而是呈阶梯状增长。因此，存在一些场景使 Prophet 和 MPC 在视频播放的全程中几乎一直选择同一种码率。在该类场景下，网络环境既不支持算法选择更高的码率，也很少在播放当前码率时发生卡顿。在这样的场景（例如丢包率为 20% 时）下，Prophet 和 MPC 的平均码率基本相同，MPC 在网络拥塞时由于其会低估下载时间（见 4.5.1 节）而倾向于保持码率不变，降低了码率切换而增大了卡顿时间；Prophet 倾向于降低码率，因此会增大码率切换并降低卡顿时间，最终导



(a) 丢包率对预测误差的影响 (b) 往返时延对预测误差的影响 (c) 带宽对预测误差的影响

Fig. 14 Comparison of prediction error for download time under different scenarios

图 14 不同场景下下载时间预测误差对比



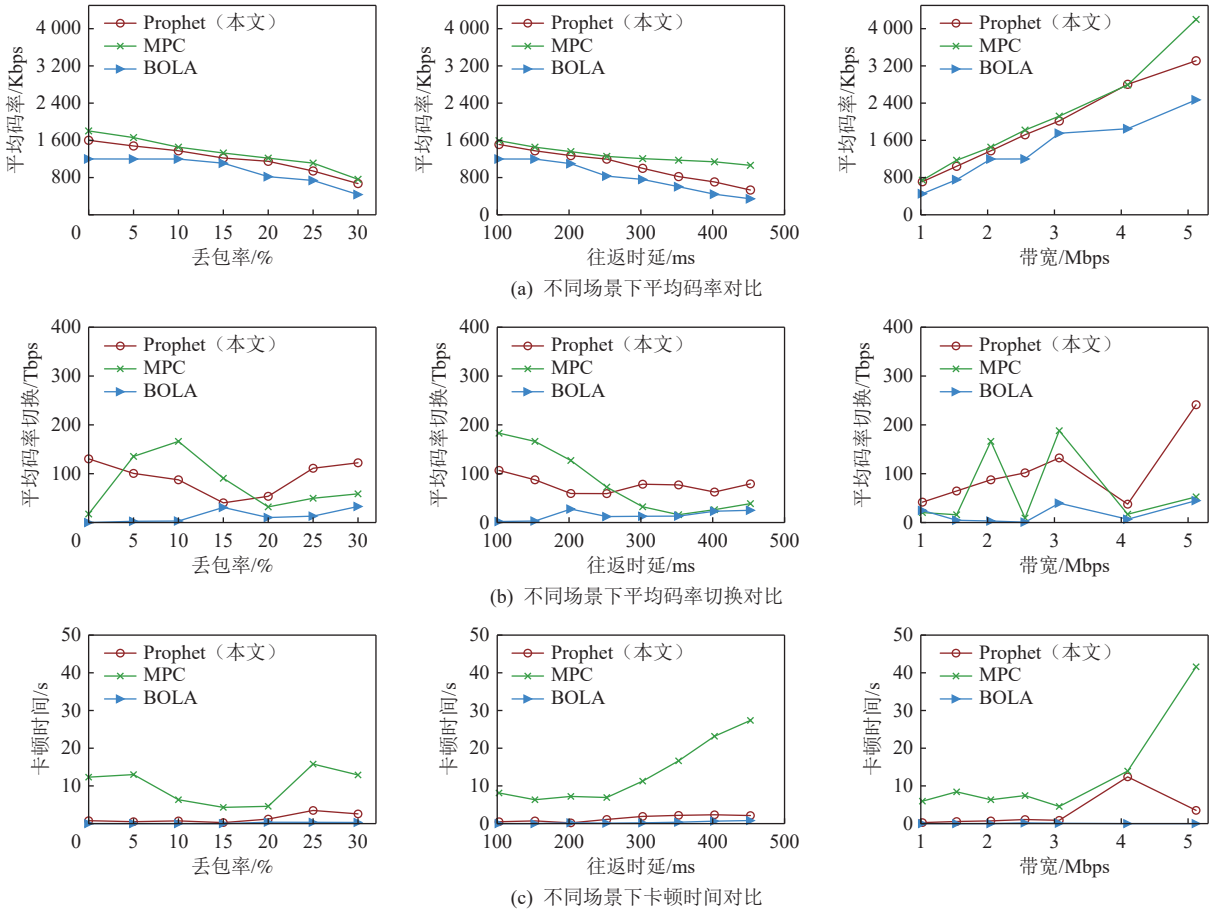
(a) 丢包率对归一化QoE的影响　(b) 往返时延对归一化QoE的影响　(c) 带宽对归一化QoE的影响

Fig. 15　Comparison of normalized QoE under different scenarios

图 15　不同场景下归一化 QoE 对比

致 2 个算法的 QoE 几乎相同。除此之外的场景，相对于 MPC，Prophet 至少能保持至少 15% 的 QoE 提升。

本实验还展示了 Prophet 和其他实验在 QoE 的各个指标上的表现。图 16 展示了 Prophet 和其他算法在 QoE 具体指标的表现，指标包括平均码率，平均码率切换和卡顿时间.Prophet 和 BOLA 的卡顿时间基本相同，但是 Prophet 的平均码率远高于 BOLA.这是由于 BOLA 必须只有达到一定的带宽阈值时才会提高码率，但是其带宽计算方法与 MPC 相似。这种在应用层测量带宽的方法往往低估带宽，导致 BOLA 在实际带宽富余的情况下仍然选择较低比特率的视频块。在不同场景下，Prophet 的平均码率比 BOLA 提高了 10.1%~59.6%，平均 QoE 比 BOLA 提高了 1.1%~54.5%。此外，Prophet 和 MPC 实现了相近的平均比特率，但是 Prophet 的卡顿时间显著低于 MPC.在不同场景下，Prophet 的卡顿时间比 MPC 降低了 73.8%~97.4%。正如 4.5.1 节所说，MPC 由于其低估了下载时间而选择高于实际网络条件的码率，因此造成了长时间的卡顿。虽然其平均码率高于 Prophet，但是其整体 QoE 仍明显低于 Prophet。



(a) 不同场景下平均码率对比

(b) 不同场景下平均码率切换对比

(c) 不同场景下卡顿时间对比

Fig. 16　Each QoE metric of different algorithms in different scenarios

图 16　不同场景下不同算法的 QoE 各项指标

综上所述, 虽然 Prophet 在每项 QoE 指标上都不优于现有最先进的算法, 但其有效平衡了各项因素, 实现了整体 QoE 的提高。

4.5.3 多场景 QoE 测试

本实验将 Prophet 算法的性能评估扩展至弱网、蜂窝网络、高动态链路、WiFi(wireless fidelity)以及有线网络 5 种典型场景, 旨在全面验证其在复杂真实网络环境中的鲁棒性和有效性。各场景设置如下:

1)弱网环境。带宽 2 Mbps, 丢包率 20%, *RTT* 为 250 ms。

2)蜂窝网络。带宽 2 Mbps, 丢包率 10%, *RTT* 为 500 ms。

3)高动态链路。带宽每隔 10 s 在 2 Mbps 到 4 Mbps 之间随机切换, 丢包率 10%, *RTT* 为 250 ms。

4)WiFi。丢包率每隔 10s 在 0.5%~3% 之间随机切换, 带宽 4 Mbps, *RTT* 为 100 ms。

5)有线网络。带宽 4 Mbps, 丢包率 0%, *RTT* 为 50 ms。

归一化 QoE 结果如图 17 所示, Prophet 在大部分场景下表现出显著的 QoE 优势。尤其在带宽波动大、丢包率高、*RTT* 长的弱网和蜂窝网络中, Prophet 的 QoE 提升更为明显。这主要得益于 Prophet 对传输层信息的精准感知, 其下载时间预测模型有效处理了丢包重传和尾部时延的影响, 从而能够更准确地判断网络状况并做出最优码率决策。图 18 中可以看出 Prophet 有效平衡了各项指标, 实现了整体 QoE 的提升。

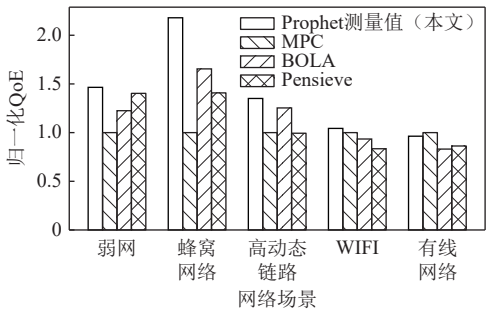


Fig. 17 Normalized QoE of various algorithms across five scenarios

图 17 各算法在 5 种场景下的归一化 QoE

在一些网络状况很好的场景下, 例如有线网络, 所有算法之间的差异并不明显, Prophet 与其他算法基本持平。原因是此时各算法都是在最高级码率和次一级码率之间来回切换, 因此差距很小。

4.5.4 带宽开销和计算开销测试

本实验测试了传输层信息的传输对于带宽开销

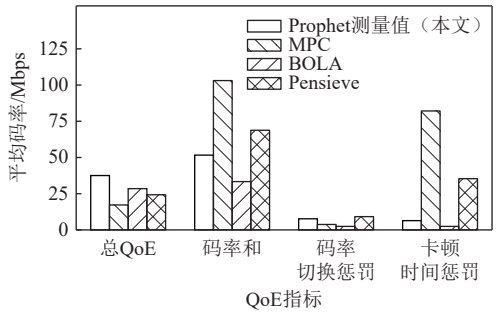


Fig. 18 Each QoE metric of various algorithms in cellular networks

图 18 蜂窝网络中各算法的 QoE 各项指标

的影响, 以及下载时间预测对于客户端计算开销的影响。

将网络环境设置为带宽 5 Mbps, 丢包 10%, *RTT* 为 250 ms, 测量客户端收到的视频块数据大小以及收到的传输层信息总大小。计算可得, 共收到视频块数据 71.57 MB, 共收到传输层信息共 116.61 KB, 传输层信息占传输数据总量的 0.159%。因此, 额外增加的传输层信息通道几乎不会对视频块的传输链路产生影响。

将网络环境设置为带宽 5 Mbps, 丢包 10%, *RTT* 为 250 ms, 测量 Prophet 和 MPC 选择码率的时间开销。2 种算法的测量结果都在 10 ms 以内。原因在于 Prophet 的下载时间预测模型的前 2 阶段计算开销与 MPC 基本一致。而第 3 阶段的计算开销对于所有码率和所有前瞻块都只需要计算 1 次, 这部分的运算时间远小于遍历码率列表和前瞻块的时间, 因为后者是随着码率级别和前瞻数量指数级上升的。因此, Prophet 的计算开销和 MPC 基本一致。

5 结 论

本文提出了一种基于传输层信息的视频码率自适应算法 Prophet。与现有主要依赖应用层信息的 ABR 算法不同, Prophet 算法利用传输层获取网络参数, 显著提升了对网络状况的实时评估能力。借助这些更精确的网络信息, 本文对于视频块的下载过程进行建模, 综合考虑了丢包重传和尾部时延等因素, 从而实现了对视频块下载时间的准确预测, 优化了码率决策。这一改进不仅有助于提高视频质量, 还有效减少视频质量的波动和播放卡顿。与现有算法相比, Prophet 算法在各种网络环境中都表现出更优秀的性能, 尤其在弱网环境下, Prophet 的优势尤为显著。

作者贡献声明：陈方舟负责方案整体设计、实现和实验验证，参与撰写论文；王清楠负责部分实验验证；单丹枫提出算法思路和实验方案，参与论文撰写和修改。

参 考 文 献

[1] Huang Tianchi, Zhou Chao, Zhang Ruixiao, et al. Comyco: Quality-aware adaptive video streaming via imitation learning[C]// Proc of the 27th ACM Int Conf on Multimedia. New York: ACM, 2019: 429−437

[2] Fortune Business Insights. Video streaming market size, share & industry analysis [EB/OL]. [2024-10-22]. https://www.fortunebusinessinsights.com/video-streaming-market-103057

[3] Apple. Apple's HTTP live streaming[EB/OL]. 2024 [2024-10-22]. https://developer.apple.com/streaming/

[4] Sodagar I. The mpeg-dash standard for multimedia streaming over the internet[J]. IEEE Multimedia, 2011, 18(4): 62−67

[5] Dobrian F, Sekar V, Awan A, et al. Understanding the impact of video quality on user engagement[J]. ACM SIGCOMM Computer Communication Review, 2011, 41(4): 362−373

[6] Kua J, Armitage G, Branch P. A survey of rate adaptation techniques for dynamic adaptive streaming over HTTP[J]. IEEE Communications Surveys & Tutorials, 2017, 19(3): 1842−1866

[7] Jiang Junchen, Sekar V, Zhang Hui. Improving fairness, efficiency, and stability in http-based adaptive video streaming with FESTIVE[C]// Proc of the 8th Int Conf on Emerging Networking Experiments and Technologies. New York: ACM, 2012: 97-108

[8] Huang Teyuan, Handigol N, Heller B, et al. Confused, timid, and unstable: Picking a video streaming rate is hard[C]// Proc of the 12th Internet Measurement Conf. New York: ACM, 2012: 225−238

[9] Li Zhi, Zhu Xiaoqing, Gahm J, et al. Probe and adapt: Rate adaptation for HTTP video streaming at scale[J]. IEEE Journal on Selected Areas in Communications, 2014, 32(4): 719−733

[10] Sun Yi, Yin Xiaoqi, Jiang Junchen, et al. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction[C]// Proc of the 30th ACM SIGCOMM Conf. New York: ACM, 2016: 272−285

[11] Huang Teyuan, Johari R, McKeown N, et al. A buffer-based approach to rate adaptation: Evidence from a large video streaming service[C]// Proc of the 28th ACM SIGCOMM Conf. New York: ACM, 2014: 187−198

[12] Spiteri K, Urgaonkar R, Sitaraman R K. BOLA: Near-optimal bitrate adaptation for online videos[J]. IEEE/ACM Transactions on Networking, 2020, 28(4): 1698−1711

[13] Yin Xiaoqi, Jindal A, Sekar V, et al. A control-theoretic approach for dynamic adaptive video streaming over HTTP[C]// Proc of the 29th ACM SIGCOMM Conf. New York: ACM, 2015: 325−338

[14] Mao Hongzi, Netravali R, Alizadeh M. Neural adaptive video streaming with Pensieve[C]// Proc of the 31st ACM SIGCOMM Conf. New York: ACM, 2017: 197−210

[15] Akhtar Z, Nam Y S, Govindan R, et al. Oboe: Auto-tuning video ABR algorithms to network conditions[C]// Proc of the 32nd ACM SIGCOMM Conf. New York: ACM, 2018: 44−58

[16] Yan F Y, Ayers H, Zhu Chenzhi, et al. Learning in situ: A randomized experiment in video streaming[C]// Proc of 17th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2020: 495−511

[17] Zhou Chao, Ban Yixuan, Zhao Yangchao, et al. PDAS: Probability-driven adaptive streaming for short video[C]// Proc of the 30th ACM Int Conf on Multimedia. New York: ACM, 2022: 7021−7025

[18] Li Yueheng, Zheng Qianyuan, Zhang Zicheng, et al. Improving ABR performance for short video streaming using multi-agent reinforcement learning with expert guidance[C]// Proc of the 33rd Workshop on Network and Operating System Support for Digital Audio and Video. New York: ACM, 2023: 58−64

[19] Iyengar J, Swett I. RFC 9002: QUIC Loss Detection and Congestion Control [S/OL]. Fremont: Internet Engineering Task Force, 2021 [2024-10-22]. https://datatracker.ietf.org/doc/html/rfc9002

[20] Cheng Yuchung, Cardwell N, Dukkipati N, et al. RFC 8985: The RACK-TLP Loss Detection Algorithm for TCP[S/OL]. Fremont: Internet Engineering Task Force, 2021 [2024-10-22]. https://datatracker.ietf.org/doc/html/rfc8985

[21] DASH Industry Forum. dash. js[EB/OL]. [2024-10-22]. https://github.com/Dash-Industry-Forum/dash.js.

[22] Bentaleb A, Taani B, Begen A C, et al. A survey on bitrate adaptation schemes for streaming media over HTTP[J]. IEEE Communications Surveys & Tutorials, 2018, 21(1): 562−585

[23] Yi Ling, Li Zeping. Research of adaptive bitrate algorithm based on deep reinforcement learning[J]. Acta Electronica Sinica, 2022, 50(5): 1192−1200(in Chinese)
(易令, 李泽平. 基于深度强化学习的码率自适应算法研究[J]. 电子学报, 2022, 50(5): 1192−1200)

[24] Wang Bo, Zhang Yuan, Yang Yongbei. Study on adaptive bitrate algorithm in decision tree based on imitation learning[J]. Computer Engineering, 2023, 49(5): 206−214 (in Chinese)
(王博, 张远, 杨咏蓓. 基于模仿学习的决策树码率自适应算法研究[J]. 计算机工程, 2023, 49(5): 206−214)

[25] Huang Tianchi, Li Chaoyang, Zhang Ruixiao, et al. A data free distillation framework for adaptive bitrate algorithm[J]. Chinese Journal of Computers, 2024, 47(1): 113−130 (in Chinese)
(黄天驰, 李朝阳, 张睿霄, 等. 决策树码率自适应算法的无数据蒸馏框架[J]. 计算机学报, 2024, 47(1): 113−130)

[26] Meng Zili, Xu Mingwei. Latency optimization in real-time multimedia transmission: Architecture, progress and the future[J]. Journal of Computer Research and Development, 2024, 61(12): 3054−3068 (in Chinese)
(孟子立, 徐明伟. 实时多媒体传输延迟优化: 架构、进展与展望[J]. 计算机研究与发展, 2024, 61(12): 3054−3068)

[27] Liu Wei, Zhang Xiaoyu, Du Wei. User allocation strategy for interactive live streaming in edge computing[J]. Journal of Computer Research and Development, 2023, 60(8): 1858−1874 (in Chinese)
(刘伟, 张骁宇, 杜薇, 等. 边缘计算中面向互动直播的用户分配策略[J]. 计算机研究与发展, 2023, 60(8): 1858−1874)

[28] Raman A, Turkkan B, Kosar T. LL-GABR: Energy efficient live video

streaming using reinforcement learning[J]. arXiv preprint, arXiv: 2402.09392, 2024

[29] Zhang Tong, Ren Fengyuan, Cheng Wenxue, et al. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in DASH[C]//Proc of the 36th IEEE Int Conf on Computer Communications. Piscataway, NJ: IEEE, 2017: 1−9

[30] Herbots J, Wijnants M, Lamotte W, et al. Cross-layer metrics sharing for QUICker video streaming[C]// Proc of the 16th Int Conf on Emerging Networking Experiments and Technologies. New York: ACM, 2020: 542−543

[31] Blanton E, Paxson V, Allman M. RFC 5681: TCP Congestion Control[S/OL]. Fremont: Internet Engineering Task Force, 2009 [2024-10-22]. https://datatracker.ietf.org/doc/html/rfc5681

[32] Blanton J, Hurtig P, Ayesta U, et al. RFC 5827: Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)[S/OL]. Fremont: Internet Engineering Task Force, 2010 [2024-10-22]. https://datatracker.ietf.org/doc/html/rfc5827

[33] Mathis M, Mahdavi J. Forward acknowledgement: Refining TCP congestion control[J]. ACM SIGCOMM Computer Communication Review, 1996, 26(4): 281−291

[34] Blanton E, Allman M, Wang Lili, et al. RFC 6675: A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP[S/OL]. Fremont: Internet Engineering Task Force, 2012 [2024-10-22]. https://datatracker.ietf.org/doc/html/rfc6675

[35] Alibaba. XQUIC[EB/OL]. [2024-10-22]. https://github.com/alibaba/xquic

[36] Taobao. Tengine[EB/OL]. [2024-10-22]. https://tengine.taobao.org/

[37] DASH Industry Forum. dash. js javascript reference client[EB/OL]. [2024-10-22].https://reference.dashif.org/dash.js/v4.7.4/samples/dash-if-reference-player/index.html

[38] Li Yueheng, Chen Hao, Xu Bowei, et al. Improving adaptive real-time video communication via cross-layer optimization[J]. IEEE Transactions on Multimedia, 2023, 26: 5369−5382

[39] hongzimao.Pensieve[EB/OL].[2025-07-08].https://github.com/hongzimao/pensieve

[40] Cardwell N, Cheng Yuchung, Gunn C S, et al. BBR: Congestion-based congestion control[J]. Communications of the ACM, 2017, 60(2): 58−66

Chen Fangzhou, born in 2001. Master candidate. His main research interest includes video transmission optimization.

陈方舟, 2001 年生。硕士研究生。主要研究方向为视频传输优化。

Wang Qingnan, born in 2001. Master candidate. Her main research interest includes network congestion control.

王清楠, 2001 年生。硕士研究生。主要研究方向为网络拥塞控制。

Shan Danfeng, born in 1991. PhD, associate professor, PhD supervisor. Member of CCF. His main research interests include network transmission control and network traffic management.

单丹枫, 1991 年生。博士，副教授，博士生导师。CCF 会员。主要研究方向为网络传输控制、网络流量管理。