Adaptive Bitrate Algorithms for Application Layer DASH Streaming: A Comparative Study and Performance Insights

Tongzhou Qian

Abstract

This paper presents a comparative analysis of three Adaptive Bitrate (ABR) algorithms—Throughput-based, BOLA, and a hybrid Dynamic strategy—within the context of Application Layer DASH Streaming using dash.js. By periodically collecting buffer, throughput, and latency metrics under alternating network constraints (switching between unthrottled and Fast 3G), we examine each algorithm's capacity to handle rapid bandwidth variations. Our findings reveal that while each algorithm has distinct strengths—whether in responsiveness, stability—they also exhibit specific shortcomings. We conclude by suggesting ways to refine these algorithms, such as more advanced bandwidth forecasting and multi-criteria optimizations, aiming to strike an optimal balance between consistency and adaptability in ABR streaming.

1 Introduction

With the continual surge in video content consumption, Dynamic Adaptive Streaming over HTTP (DASH) has grown indispensable in delivering seamless Video on Demand (VoD) experiences. DASH breaks video content into smaller segments, or *streamlets*, and allows the client to decide which bitrate to fetch based on real-time network conditions. The goal is to leverage available bandwidth effectively while minimizing buffering and playback disruptions.

In this paper, we explore three Adaptive Bitrate (ABR) methods commonly encountered in DASH implementations:

- **Throughput-based:** Selects the next segment's quality based on recently measured bandwidth, enabling rapid scaling in favorable network conditions but risking abrupt quality downgrades during sharp bandwidth drops.
- **BOLA (Buffer Occupancy based Lyapunov Algorithm):** Adapts segment quality primarily according to the current buffer occupancy, ensuring a steady buffer but sometimes reacting slowly to sudden bandwidth improvements.
- **Dynamic:** Combines both buffer-based and throughput-based strategies, aiming to maintain a balance between quick adaptation to network changes and steady playback quality.

The aim of this study is to evaluate how these ABR algorithms respond to substantial network fluctuations by imposing alternating bandwidth constraints during playback (one minute of no throttling followed by two minutes of Fast 3G, repeated). We collected data at eight-second intervals, focusing on how quickly and effectively each ABR method adjusts to maintain video quality and minimize rebuffering under abrupt network changes.

The remainder of this paper is structured as follows. Section 2 outlines our experimental setup, including data collection and network configurations. Section 3 summarizes our key findings, highlighting each algorithm's trade-offs in terms of buffer stability, throughput accuracy, and latency. Finally, Section 4 offers conclusions and discusses future work, including potential areas for refining bandwidth estimation and QoE-aware decision making.

2 Experimental Setup and Methodology

2.1 Metrics Collection

To capture the real-time performance of the three ABR algorithms, we employed a JavaScript snippet that retrieves key metrics every eight seconds. We used the `player.getDashMetrics()` and `player.getAverageThroughput()` APIs in dash.js to obtain:

- **Buffer Level** (seconds)
- **Throughput** (kbps)
- **Latency** (seconds)

Below is a condensed version of the JavaScript code, illustrating how we record these values and print them to the console for further analysis:

```
setInterval(() => {
  let videoMetrics = player.
      getDashMetrics();
  let bufferLevelSec = "N/A",
      avgThroughputKbps = "N/A",
      avgLatencySec = "N/A";

  if (videoMetrics) {
    // Retrieve current buffer (in
        seconds).
    const currentBufferLevel =
      videoMetrics.getCurrentBufferLevel
          ("video");
    if (currentBufferLevel !== null) {
      bufferLevelSec = currentBufferLevel
          .toFixed(2);
    }

    // Fetch average throughput (in kbps)
        .
    const throughputKbps =
      player.getAverageThroughput("video
          ");
    if (throughputKbps) {
      avgThroughputKbps = throughputKbps.
          toFixed(2);
    }

    // Calculate average latency from
        recent requests.
    const httpRequests = videoMetrics.
        getHttpRequests("video");
    if (httpRequests && httpRequests.
        length > 0) {
      let latencyValues = [];
      for (let j = httpRequests.length -
          1;
               j >= 0 && latencyValues.
                   length < 4; j--) {
        let req = httpRequests[j];
        if (req.trequest && req.tresponse
            ) {
          let startTime = new Date(req.
              trequest).getTime();
          let endTime = new Date(req.
              tresponse).getTime();
          latencyValues.push(endTime -
              startTime);
        }
      }
      if (latencyValues.length > 0) {
        let sumLatency = latencyValues.
            reduce(
          (sum, val) => sum + val, 0);
        avgLatencySec = (sumLatency /
          latencyValues.length / 1000).
              toFixed(2);
      }
    }
  }

  // Log the collected metrics
  const logMessage = `Buffer: ${
      bufferLevelSec}s, ` +
    `Throughput: ${avgThroughputKbps}kbps
        , ` +
    `Latency: ${avgLatencySec}s`;
  console.log(logMessage);
}, 8000);
```

Figure 1 shows an example of the log output in the browser console.

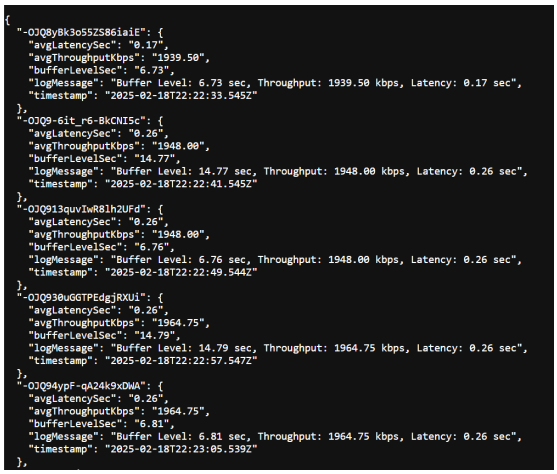

Figure 1: Console log of collected metrics

2.2 Data Processing and Analysis

The raw data was subsequently processed to transform and visualize the metrics. A PHP script (shown in abbreviated form below) was utilized to parse JSON logs, convert numerical values, and group them for plotting time-series graphs. During this step, values like "`10s`" or "`1500 kbps`" were standardized into numeric formats.

```
$dataArray = json_decode($jsonData, true)
    ;

// Prepare arrays for charting
$timestamps = [];
$bufferLevels = [];
$throughputs = [];
$latencies = [];

foreach ($dataArray as $id => $entry) {
    // Format or convert entries
    $timestamps[] = date("H:i:s",
        strtotime($entry["timestamp"]));
    $bufferLevels[] = (float)$entry["
        bufferLevelSec"];
    $throughputs[] = (float)$entry["
        avgThroughputKbps"];
    $latencies[] = ($entry["avgLatencySec
        "] === "N/A")
        ? null
        : (float)$entry["avgLatencySec"];
}
```

These arrays were then used to plot time-based charts, enabling an intuitive comparison of buffer levels, throughput, and latency for each ABR algorithm.

2.3 Network Configuration

To reproduce real-world scenarios, we applied alternating bandwidth constraints in Microsoft Edge:

- **No Throttling:** 1 minute of unconstrained bandwidth.
  - Download: $\sim$1900 kbps
  - Latency: $\sim$0.26 s
- **Fast 3G:** 2 minutes of throttled bandwidth, approximating:
  - Download: $\sim$600 kbps
  - Latency: $\sim$1.5 s

We cycled between these two configurations twice in each playback session, allowing us to observe how quickly each algorithm recovered or adjusted when the available bandwidth shifted. The Fast 3G profile was chosen to emulate a moderately constrained mobile or cellular environment.

2.4 Potential Limitations

Although our experimental design attempts to be systematic, we acknowledge a few limitations:

- **Transition Phases:** Data points captured right at the switch between No Throttling and Fast 3G might not reflect pure conditions of either state.
- **Browser Delays:** The precise moment when Chrome applies or removes throttling could introduce slight inconsistencies.
- **Initialization Effects:** Early measurements may be affected by player startup, skewing buffer and latency metrics.
- **External Network Variability:** Even with throttling, baseline network fluctuations can occur, resulting in occasional outliers.

Nevertheless, the collected dataset provides a sound basis for evaluating each ABR approach under realistic, though controlled, conditions.

3 Results and Discussion

In this section, we highlight significant observations drawn from the processed results. Figures 2, 3, 4, and 5 illustrate each algorithm's behavior in terms of buffer occupancy, throughput, and latency, as well as an overarching comparison.

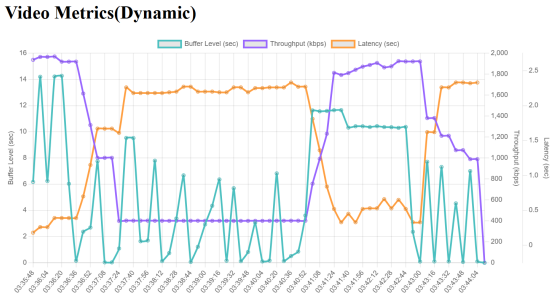

Figure 2: Performance of the Dynamic ABR Algorithm

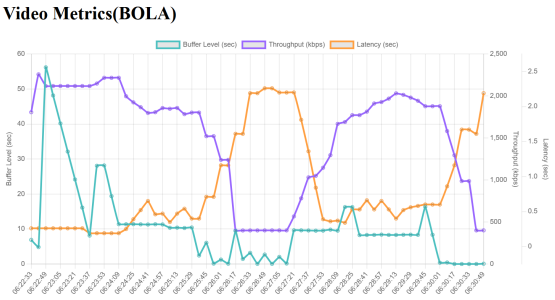

Figure 3: Performance of the BOLA Algorithm

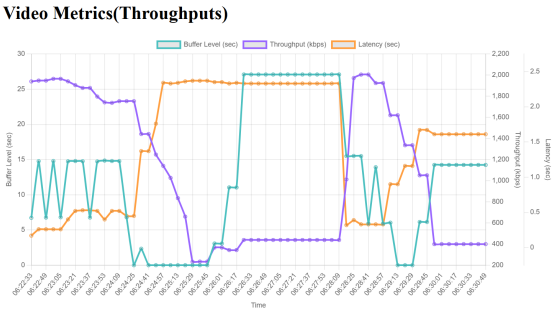

Figure 4: Performance of the Throughput-based Algorithm

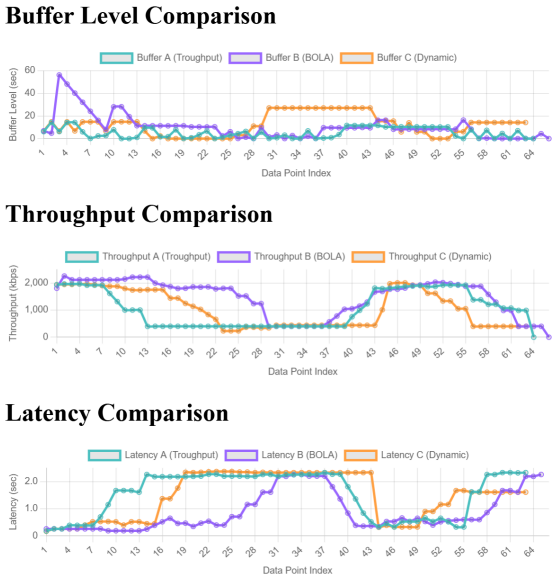

Figure 5: Cross-Algorithm Comparison: Buffer Level, Throughput, and Latency

3.1 Buffer Stability

- **Throughput-based:** Tends to accumulate buffer quickly under high bandwidth but may suffer precipitous drops when throttling kicks in before it recalculates the available bandwidth.
- **BOLA:** Generally exhibits steadier buffer occupancy but lags when bandwidth spikes, potentially underutilizing the newfound capacity.

- **Dynamic:** Shifts between these strategies, maintaining a moderate buffer range (often 10–15 seconds) to mitigate abrupt drops.

3.2 Throughput Utilization

- **Throughput-based:** Quickly capitalizes on elevated bandwidth, but can overshoot during steep declines in network performance.
- **BOLA:** Less aggressive, leaning toward a more stable buffer and possibly not leveraging peak bandwidth as much.
- **Dynamic:** Achieves a middle ground, using available bandwidth effectively while avoiding severe bitrate fluctuations.

3.3 Latency Patterns

- **Throughput-based:** Typically low under stable bandwidth, yet more prone to spikes when throttling is applied.
- **BOLA:** Operates on the buffer margin to stabilize playback, often with moderately varying latency.
- **Dynamic:** Reactive switching helps maintain moderate latency levels, but small spikes can still occur during abrupt bandwidth changes.

3.4 Overall Observations

Throughput-based approaches offer quick adaptation but risk instability under erratic networks. BOLA emphasizes consistency, especially with buffer management, though it may react slowly to sudden bandwidth improvements. The Dynamic approach blends both, aiming to optimize for rapidly shifting conditions by strategically switching between throughput- and buffer-based logic.

In practice, achieving the right trade-off depends heavily on application-specific needs. If minimizing rebuffering is critical (e.g., for live sports), a slightly more conservative, buffer-centric scheme may be preferable. Conversely, if maximizing video quality is a priority, a throughput-focused approach might be suitable, provided it incorporates safeguards against sudden bandwidth dips.

4 Conclusion

This study assessed three ABR algorithms—Throughput-based, BOLA, and a hybrid Dynamic solution—under alternating network conditions for Video on Demand streaming. Our experiments show that each algorithm excels in certain scenarios:

- **Throughput-based:** Rapid adaptation to high bandwidth, with potential for abrupt quality drops under throttling.
- **BOLA:** Steady buffer management and reduced rebuffering, at the cost of sometimes missing out on brief bandwidth surges.
- **Dynamic:** Balances fast adaptation with buffer stability, toggling between throughput- and buffer-awareness based on real-time metrics.

While each technique provides distinct advantages, further enhancements are possible. For instance, incorporating advanced bandwidth prediction (e.g., machine learning or history-based models) could smooth out abrupt fluctuations. Ultimately, the optimal choice of ABR algorithm should align with a service's primary objectives, whether that be seamless user experience, maximum video quality, or a balanced compromise.

References

- http://qiantongzhou.huizhoutech.top/projects-/comp6461-lab2-diagram.php/Data Collection Server
- https://netflixtechblog.com/a-closer-look-at-netflixs-abr-algorithms-7b8f1b5b3b2Netflix Tech Blog: A closer look at Netflix's ABR algorithms
- https://cdn.dashjs.org/latest/jsdoc/module-MediaPlayer.htmldash.js MediaPlayer API Documentation