Public Review for

Foresight: Planning for Spatial and Temporal Variations in Bandwidth for Streaming Services on Mobile Devices

Manasvini Sethuraman, Anirudh Sarma, Ashutosh Dhekne, Umakishore Ramachandran

The paper develops Foresight to provide an up-to-date improvement of earlier ideas for the prediction of bandwidth availability for arbitrary applications on mobile devices, based on CRUSP's crowdsourced reporting to a geographical database and application-independent querying by mobile devices. The paper uses video streaming as an example and comprises an actual implementation using a video player based on ExoPlayer.

Compared to earlier work, the papers relies on recently collected data, which improves the realism of the bandwidth prediction in today's mobile space and implicitly takes development in mobile networks into account. The application-independent provision of geo-located bandwidth lookup is a key feature proposed by the paper, which improves privacy by limiting access to the user's location to one service provider rather than arbitrary application providers.

The reviewers liked the inspiration that the paper provides and appreciate the work that the authors have put into the development of a functional system. The reviewers would have preferred if the authors had either performed a comparison with the earlier work on crowdsourcing-based predictive streaming or, alternatively, explained the value of the knapsack algorithm meant to facilitate sharing between applications by demonstrating Foresight with several applications.

Public review written by
Carsten Griwodz
University of Oslo, Norway

Foresight: Planning for Spatial and Temporal Variations in Bandwidth for Streaming Services on Mobile Devices

Manasvini Sethuraman
msethuraman3@gatech.edu
Georgia Institute of Technology

Anirudh Sarma
anirudhs@gatech.edu
Georgia Institute of Technology

Ashutosh Dhekne
dhekne@gatech.edu
Georgia Institute of Technology

Umakishore Ramachandran
rama@gatech.edu
Georgia Institute of Technology

Abstract

Spatiotemporal variation in cellular bandwidth availability is well-known and could affect a mobile user's quality of experience (QoE), especially while using bandwidth intensive streaming applications such as movies, podcasts, and music videos during commute. If such variations are made available to a streaming service in advance it could perhaps plan better to avoid sub-optimal performance while the user travels through regions of low bandwidth availability. The intuition is that such future knowledge could be used to buffer additional content in regions of higher bandwidth availability to tide over the deficits in regions of low bandwidth availability. *Foresight* is a service designed to provide this future knowledge for client apps running on a mobile device. It comprises three components: (a) a crowd-sourced bandwidth estimate reporting facility, (b) an on-cloud bandwidth service that records the spatiotemporal variations in bandwidth and serves queries for bandwidth availability from mobile users, and (c) an on-device bandwidth manager that caters to the bandwidth requirements from client apps by providing them with bandwidth allocation schedules. Foresight is implemented in the Android framework. As a proof of concept for using this service, we have modified an open-source video player—Exoplayer—to use the results of Foresight in its video buffer management. Our performance evaluation shows Foresight's scalability. We also showcase the opportunity that Foresight offers to ExoPlayer to enhance video quality of experience (QoE) despite spatiotemporal bandwidth variations for metrics such as overall higher bitrate of playback, reduction in number of bitrate switches, and reduction in the number of stalls during video playback.

CCS Concepts

• **Networks → Location based services**; • **Information systems → Multimedia streaming**.

MMSys 21, September 28-October 1, 2021, Istanbul, Turkey

ACM ISBN 978-1-4503-8434-6/21/09…$15.00
https://doi.org/10.1145/3458305.3463384

Keywords

Spatiotemporal Bandwidth Information, Bandwidth Management

ACM Reference Format:

Manasvini Sethuraman, Anirudh Sarma, Ashutosh Dhekne, and Umakishore Ramachandran. 2021. Foresight: Planning for Spatial and Temporal Variations in Bandwidth for Streaming Services on Mobile Devices. In *12th ACM Multimedia Systems Conference (MMSys '21) (MMSys 21), September 28-October 1, 2021, Istanbul, Turkey*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458305.3463384

1 Introduction

Interruptions to streaming services (*e.g.,* video/audio/podcast) are a common occurrence when using mobile devices on the move. In most cases the culprit is intermittent connectivity as a user travels across multiple cellular regions and sometimes through regions of low cellular coverage. As cellular connectivity transitions to 5G, due to inherent fading characteristics of mmWave signals, such spatial variations are expected to increase even further [17, 28]. If the spatiotemporal bandwidth variations are made available *a priori* to a streaming service, it could plan ahead on buffering more content in preparation for traveling through an area of low bandwidth. Given that there are multiple streaming apps on a mobile device that could benefit from such prior knowledge, it would be prudent to make the knowledge of spatiotemporal bandwidth availability a *first class citizen* on mobile devices.

Variations in bandwidth availability is well-known. While urban and densely populated cities are expected to have reliable mobile connectivity, Fig. 1 shows that the there are significant spatial variations in available bandwidth for the cities of New York, Chicago, and Atlanta.[1] It is evident that many poor bandwidth locations abut other locations with high bandwidth. When a user travels through these low bandwidth areas, stalls or quality reductions in streaming services are likely to occur. In addition to spatial variations, mobile users also experience temporal variations in bandwidth for a given location [30]. The default *reactive* action by a streaming service such as a video player is to reduce the bitrate upon noticing a drop in available bandwidth below a threshold. If a streaming service could *know* that a low bandwidth area is imminent, it can take *proactive* action by fetching additional data beforehand. Hence, if sufficient video data has already been buffered, intermittent low

[1]The figure was built from Ookla dataset[25] and US government census data [4] for city and county boundaries.

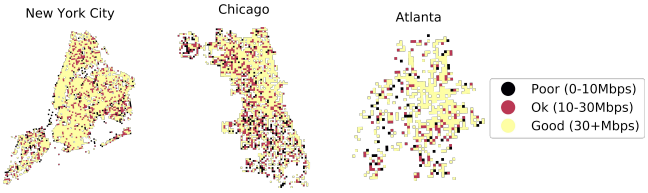

Figure 1: Spatial variation in available bandwidth from Ookla dataset

bandwidth areas can be simply ignored since the mobile device is expected to re-enter a high bandwidth area soon.

There is prior work that uses spatiotemporal bandwidth variations to adapt the bitrate of a video player built in the application itself [14]. Another work explicitly uses the response time for segment acquisition in a video player to adapt the bitrate [33]. While a detailed comparison of our work to the related work is presented in §6, it suffices to say that, to the best of our knowledge, no prior art provides spatiotemporal bandwidth forecasting as a service on a mobile device for use by any app on the device in a location-obfuscated manner. Such location obfuscation provides bandwidth updates without revealing users' whereabouts to individual apps.

In this work, we elevate *location-obfuscated* forecasting of *available bandwidth* over time to a first class citizen in a mobile device that is available for any app on the device, to account for the spatiotemporal bandwidth variations.

Foresight has two parts: an on-cloud *bandwidth service*, and an on-device *scheduling service*. The first part is a light-weight crowd-sourced mechanism for uploading and querying spatiotemporal bandwidth information to a cloud-resident bandwidth service. The scheduling service on the device queries the bandwidth service in the cloud to get a snapshot of the available bandwidth along the spatial dimension, based on the user's selected route. It uses this information to generate the bandwidth schedule along the time axis for client apps on the device that query the scheduling service. There are two important aspects to the novelty of work. First, a user may have multiple apps on their device (*e.g.,* Netflix, YouTube, Spotify, apps performing sync operations in the background *etc.*). Elevating such a service to the device level obviates the need for bandwidth forecasting at the individual apps level. Second, the apps do not need access to sensitive information (*e.g.,* the location of the user nor the route being taken by the user) to know the future bandwidth availability.

Foresight is dependent on reliable data from a good bandwidth estimation service. However, the impact on the mobile device for participating in the data collection should be minimal. Popular bandwidth estimation services, such as speedtest.net [26] and fast.com [22] operate by incrementally clogging the link to a nearby server from the mobile device. Such an approach imposes a huge overhead on a user's mobile data. Instead, we make use of a light-weight probing tool called CRUSP [31] to approximate the available bandwidth for a location. This tool periodically collects bandwidth estimates over LTE with a small overhead (less than 2MB of data transmission per bandwidth estimation). We envision a bandwidth estimator based on CRUSP (possibly bundled in with Foresight) to be running continuously on client devices to populate the on-cloud bandwidth service with crowd-sourced data.

An app on a mobile device will contact the on-device bandwidth scheduling service with its bandwidth requirement (*e.g.,* "need 100 Mbps"). The forecasting service gets the route the mobile user is traveling on, from a mapping service such as Google Maps on the device. Foresight then contacts its cloud counterpart to get the bandwidth information along the route. This information is updated occasionally to allow for temporal variations as well as for route changes. The on-device bandwidth scheduling service will use this information to arrive at a schedule of available bandwidth (as a function of time) for use by the app (*e.g.,* 90 Mbps for 10 seconds, 100 Mbps for 20 seconds, 100 Mbps for 10 seconds, *etc.*).

An app on a mobile device may use the bandwidth schedule it receives from Foresight in a manner consistent with its goals; *e.g.,* a video player may use this bandwidth schedule to reduce the number of stalls in playing a video, provide a consistent video quality without bitrate switches throughout the length of the video, or strive to provide the highest quality (*i.e.,* bitrate) video for as long as possible. The design decisions for a video player would depend on the desired goals. In this work, we are not concerned as to *how* the information from Foresight would be used in a streaming service, though we indeed show that simple modifications can yield results matching any of these goals.

Our primary contribution is Foresight with its two components:

(1) A cloud-resident bandwidth service that, given a route, responds with estimates of the available bandwidth along the spatial dimension, and the design exploration inherent in creating an efficient implementation of such a service.
(2) A device-resident bandwidth manager that provides location-obfuscated schedule of bandwidth availability as a function of time to requesting streaming apps on the mobile device. The bandwidth manager takes requests from multiple competing applications on the same device and allocates bandwidth to each using a greedy knapsack algorithm. To account for temporal variations in the bandwidth, Foresight uses *epochs* to periodically query the on-cloud bandwidth service and update the client apps on the bandwidth allocation.

Additionally, we modify an open-source video player (ExoPlayer [11]) by implementing a buffer length adaptation algorithm that uses the results from Foresight. Finally, we carry out performance evaluation to demonstrate the scalability of the cloud-resident bandwidth service, and the low-overhead of the epoch-based on-device scheduling service by the bandwidth manager. Further, using the modifications to ExoPlayer, we showcase how the bandwidth forecasting can be used to reduce stalls and achieve better user experience in terms of perceived video streaming quality and bitrate switches.

The rest of the paper is organized as follows: §2 describes Foresight's system architecture. §3 follows with implementation details of Foresight. §4 showcases how the results returned by Foresight could be used by an exemplar video player (ExoPlayer). §5 presents the evaluation of Foresight and the exemplar video player. §6 explores the prior art and tries to situate Foresightwith respect to the prior art in the domains of spatiotemporal bandwidth estimation, and techniques for incorporating such data into mobile devices. §7 discusses some of the limitations of our current work and provides suggestions for future work, followed by concluding remarks in §8.

2 System Architecture of Foresight

Foresight has two components: (1) bandwidth service in the cloud, and (2) bandwidth management in the device. The on-cloud bandwidth service maintains a spatiotemporal datastore of bandwidth availability. The on-device bandwidth management takes in requests for bandwidth allocation from client apps and acquires the mobile user's route from an on-device app (*e.g.,* Google Maps). It then queries the on-cloud bandwidth service for *en route* bandwidth availability, and creates a per-app location-obfuscated bandwidth allocation schedule along the time axis. The components of the system architecture are shown in Fig. 2. §2.1 discusses the design space exploration for the on-cloud bandwidth service; and §2.2 covers on-device bandwidth management.

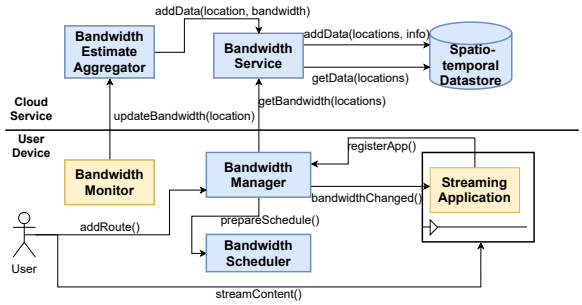

Figure 2: Foresight system architecture. Blue boxes are our contributions, yellow boxes are modifications that we made to existing applications.

2.1 On-Cloud Bandwidth Service

The cloud service (depicted in the top half of Fig. 2) maintains a *spatiotemporal datastore* and controls the insertion and retrieval of the data from this store. Users who are feeding in bandwidth data (as part of crowd-sourcing) will send locally gathered information about their experienced bandwidth, estimated through a low-overhead *bandwidth estimation pipeline* (shown in Fig. 3), which is then aggregated and fed into the spatiotemporal datastore. Queries on bandwidth availability for a set of geographical locations from client devices are fielded by the bandwidth service which returns data from the spatiotemporal datastore.

2.1.1 Spatiotemporal Datastore

The spatiotemporal datastore's function is to enable fast retrieval of bandwidth information in the vicinity of a given query location. Fast retrieval is achieved via a spatially indexed key value store that allows proximity queries to be made efficiently. GeoHash [23] is a technique that encodes a geographic location into a string, retaining the ordering of the point in space. When adding a point represented by a <*latitude, longitude*> pair, first the GeoHash function is applied to the <*latitude, longitude*> pair, producing a string that is then stored along with the original <*latitude, longitude*>. Queries for a given <*latitude, longitude*> pair are similarly transformed into a string using the same GeoHash function, thus converting a spatial search into a string search.

In order to retrieve relevant bandwidth estimates in a geographical area, we search the spatiotemporal datastore for data points that fall within a certain radius of the queried location. Since freshness

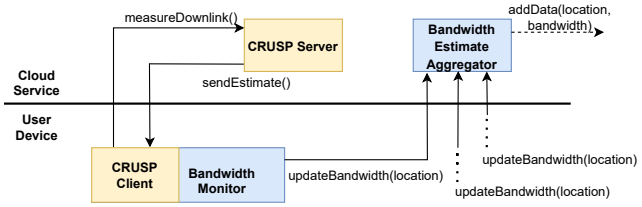

Figure 3: Bandwidth estimation pipeline

of data is important, we limit the search results to only return the moving average of bandwidth measurements reported at a location within the last hour. The indexing scheme for the datastore is as follows: we break up the geographical area into a grid and represent each unit in the grid by its GeoHash string which acts as the key into the spatiotemporal bandwidth index. The values corresponding to each key represent a set of points, along with the estimated bandwidth for each of those points.

2.1.2 Bandwidth Estimation Pipeline

To facilitate crowd-sourced acquisition of bandwidth estimates, we need a low overhead bandwidth monitoring facility that can be run on mobile devices. There are two primary considerations for a cellular bandwidth estimation tool: 1) avoid excessive use of user's mobile data; and 2) estimate bandwidth quickly and reliably. On the one hand, conventional bandwidth measurements (such as SpeedTest [26]) provide reliable estimates at the cost of significant overhead to the mobile device; and on the other hand, low-overhead packet dispersion techniques which use packet pairs and packet trains are unreliable in estimating instantaneous cellular bandwidth [37]. We found a reliable low-overhead bandwidth estimation technique in CRUSP [31], which works by exchanging a few carefully crafted probe packets that reveal the approximate bandwidth. CRUSP examines the burst of packets received, which were originally transmitted by the server at a constant rate to estimate bandwidth with frugal use of resources. Moreover, the reliability of CRUSP in live LTE networks has been extensively investigated [31]. Hence, we have adopted CRUSP into our bandwidth estimation pipeline depicted in Fig. 3.

2.2 On-Device Bandwidth Manager

The on-device bandwidth manager (bottom half of Fig. 2) accepts requests for bandwidth allocation from the client apps running on the device. It obtains the user's route information from an on-device navigation app (we built a prototype application that queries Google Maps API [12] for the route and sends the route to the bandwidth manager). The bandwidth manager queries the on-cloud bandwidth service (as shown in Fig. 2) to receive the spatial bandwidth estimates for the chosen route in the form of <*location, bw-estimate*> tuples. The bandwidth manager transforms the <*location, bw-estimate*> tuples into <*time, bw-estimate*> tuples. It does this transformation by querying an on-device service (such as Android's GPS facility) to estimate the expected arrival times at each of the location points, and then mapping the bandwidth estimates in spatial domain to corresponding values in the time domain. The bandwidth estimates and the allocation requests are then fed to the bandwidth scheduler (described in §2.2.1). The scheduler computes the per-app bandwidth allocation in the time domain,

which is then returned to each client app. There could be temporal variations in the bandwidth estimates for a given location. The bandwidth manager queries the on-cloud bandwidth service for the bandwidth estimates for the chosen route with some periodicity (termed as *epoch*). The bandwidth scheduler is invoked after each such query if there is change in the bandwidth estimates beyond a chosen threshold (described in §2.2.2). The bandwidth manager then provides updates on the allocation schedule to the registered clients if there are schedule changes due to the temporal variations in estimated bandwidth.

2.2.1 Bandwidth Scheduler

Multiple applications could make bandwidth allocation requests simultaneously to the bandwidth manager. The role of the bandwidth scheduler is to make bandwidth allocations over time for the requesting apps commensurate with the spatio-temporal variations in the available bandwidth estimates. The bandwidth requirement of each application could be different, and the way apps are used in a mobile device could be quite different. For *e.g.*, a video streaming application would require high bandwidth and run in the foreground; whereas music streaming is not bandwidth intensive and could run in the background. Intuitively, we want to allocate more bandwidth for applications that are heavily used *i.e.,* they have a high *value*. We model bandwidth allocation in terms of the Knapsack problem—the goal is to fill a knapsack of capacity W, with n items having values $v_1, v_2, \ldots v_n$ and weights $w_1, w_2, \ldots w_n$, assuming we can pick items not just their entirety, but also as fractions, then we get:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \mid \sum_{i=1}^{n} w_i x_i \leq W \text{ and } 0 \leq x_i \leq 1 \qquad (1)$$

We represent application bandwidth requirements as item weights, and the application's usage as the values. Thus, if we have applications with bandwidth requirements $r_1, r_2, r_3..r_n$, and we compute the application usage (time spent by the user on the application) as $u_1, u_2, u_3, ...u_n$, and the total bandwidth available is B, solving for the Knapsack problem would yield optimal bandwidth allocation requirements. A greedy solution to the fractional version of the Knapsack problem is to compute the ratio of v_i to w_i and sort the items in descending order of v_i/w_i, and then fill the knapsack until the total knapsack capacity is reached. The `getAllocations` function in Algorithm 1 outlines the bandwidth allocation using the greedy solution to the Knapsack problem.

Once the bandwidth manager receives estimates of bandwidth along the route from the on-cloud bandwidth service, it transforms the bandwidth estimates in space to bandwidth estimates in time. Then, the bandwidth manager calls the `createSchedule` function in Algorithm 1. The input to the scheduler is a series of <*time, bw-estimate*> values. The scheduler thus has to call the `getAllocations` function for every one of these values in the input time series of bandwidth estimates from the bandwidth manager. It is possible that the bandwidth estimates could be fairly unchanging along the route for a stretch of the journey. Therefore, the scheduler uses a *change threshold* to reduce the number of times the knapsack algorithm has to be run. Thus, if the change in bandwidth estimates from one location along the route to another is less than the *change threshold*, then we simply reuse the previous estimate (lines 21–23 in Algorithm 1),

Algorithm 1: Bandwidth Scheduler

```
1  Function getAllocations(appInfo, totalBw):
2      allocatedBandwidth = 0;
3      appAllocation = {};
4      for (app: appInfo) do
5          app.usageToRequirement = app.usage /
             app.requirement;
6          app.allocation = 0;
7      end
8      sort(appInfo, by=usageToRequirement);
9      while allocatedBandwidth ≤ totalBw do
10         allocation = min(app.requirement, totalBanwidth -
             allocatedBandwidth);
11         allocatedBandwidth += allocation;
12         appAllocation.put(app.name, allocation);
13     end
14     return appAllocation;
15 Function
     createSchedule(appInfo, bwInfo, changeThreshold):
16     // Called by the Bandwidth manager upon receiving
         bandwidth estimates
17     bwSchedule = [];
18     oldBw = 0;
19     allocations = [];
20     for (i=0; i< bwInfo.size; ++i) do
21         if (i ==0 or bwInfo[i].bw - oldBw > changeThreshold)
             then
22             allocations = getAllocations(appInfo, bwInfo.bw);
23         end
24         bwSchdule.put(bwInfo[i].duration, allocations);
25         oldBw = bwInfo[i].bw;
26     end
27     return bwSchedule;
```

skipping a call to the `getAllocations` function for such sequence of locations. We explore the implications of having such thresholds on the scheduler's performance in §5.

2.2.2 Epoch-based Refresh of Allocation Schedule

The bandwidth manager that fetches the entire route's bandwidth estimates from the on-cloud bandwidth service must decide:

(1) How often should the bandwidth estimates be refreshed?

(2) How often should the bandwidth allocations be re-computed?

The first question pertains to the temporal variations in the bandwidth estimates along the route as reported via crowd-sourcing to the on-cloud bandwidth service. The second question pertains to the extent of the temporal variations that may or may not warrant re-calculating the allocation recommendations made to the client apps. To answer the first question, we introduce *epochs*, which control the frequency with which the bandwidth estimates are refreshed by querying the cloud server. We do not know beforehand how frequently the bandwidth measurements are updated on the cloud server. Hence, we explore two strategies for refreshing the bandwidth estimates: (1) *periodic epochs* — querying the

cloud server at fixed intervals; and (2) *adaptive epochs* – adaptively changing the querying interval based on the magnitude of change in the bandwidth estimates. While the first strategy is simple to implement, in the long run it could potentially waste network resources if the bandwidth measurements change infrequently and the epoch is too small, or if the epoch size is too large, changes to available bandwidth measurements might be missed. The adaptive scheme, similar to congestion control in TCP, gradually increases the epoch duration if significant changes are not observed in the bandwidth estimates, but halves the epoch duration if significant changes are observed. The second question is addressed using the *change threshold*, which we introduced in §2.2.1. At the beginning of each epoch, on receiving a fresh set of bandwidth estimates, the bandwidth manager calls the top-level function in the scheduler (`createSchedule` in Algorithm 1). Once an initial schedule has been created, `createSchedule` calls `getAllocations` for a specific location *only* if the change in the bandwidth estimate for that location exceeds the threshold. The newly computed bandwidth allocation is communicated to the client apps as an update to the initial schedule. The two parameters – epoch duration and change threshold – help us in evaluating the trade-offs between network communication, scheduling overhead, and the accuracy of the allocation recommendations by the scheduler. We explore the effect of both of these parameters in §5.

3 Implementation of Foresight

In this section, we present the technology choices and implementation details of Foresight.

3.1 On-Cloud Bandwidth Service

The bandwidth service comprises two components: (1) the bandwidth estimation pipeline, and (2) the spatiotemporal datastore.

3.1.1 Crowd Sourced Bandwidth Estimation Pipeline

The bandwidth estimation pipeline consists of the CRUSP client and CRUSP server, derived from the publicly available original implementation [15]. The CRUSP client is integrated into the *bandwidth monitor*, which is the Android app used for crowd-sourcing (Fig. 3). The app aggregates bandwidth measurements on the device and periodically reports the bandwidth estimates to the bandwidth estimate aggregator (in the cloud) via HTTP post requests. Loss of connectivity to the cloud, *e.g.,* when a user travels through a tunnel, is also important information. Such zero-estimates are recorded along with the last known location, and are batch-uploaded to the cloud server when connection is reestablished.

3.1.2 Spatiotemporal Datastore

Our requirements for the spatiotemporal datastore included quick access to live data, ease of use, scalability, and durability. For all these reasons we chose Redis for our implementation.

For indexing spatial information, Redis uses a data structure called Z-set which preserves the locality of data points. We expose the spatiotemporal bandwidth information via a Java service built on the Jetty Framework [8], which is lightweight and scalable; Lettuce library [18] is used to access Redis via asynchronous I/O.

Bandwidth estimation data points added to the key-value store are represented as <*key, latitude, longitude, value*> tuples, where *value* is the reported bandwidth estimate. Each key maps to a set of

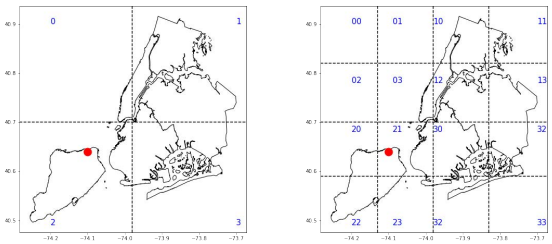

Figure 4: GeoHash representation of points. (a) 1-character representation: the red dot is hashed to string '2' (b) 2-character representation: the red dot is hashed to string '21'.

<*latitude, longitude, value*> tuples. It is necessary to choose the key representation carefully because, for any given point we want to determine quickly which key we need to look up, and then perform a spatial search within the points corresponding to the key. We want to avoid accessing multiple keys for a single location.

We found GeoHash to be a suitable candidate for representing the keyspace. For a given latitude/longitude pair, GeoHash outputs a string representation of the point. When a point is added as <*latitude, longitude, value*>, we first compute the GeoHash of the <*latitude, longitude*> pair, and then store it in Redis as <*GeoHash(latitude, longitude, n), latitude, longitude, value*>. Here, *n* is the number of characters used in producing the GeoHash string. When we increase the number of characters in the GeoHash, we increase the accuracy with which the point is represented.

Thus, in choosing the number of characters to represent the key, we limit the size of the area represented by the key. For example, we could use a single character representation for the geographical area as shown in Fig. 4(a); or we could break up the geographical area into a more granular grid as shown in Fig. 4(b). With a single character GeoHash, the red dot in Fig. 4(a) would be represented by the character "2". But so would all the other locations in the same lower left quadrant as the red dot. On the other hand, in Fig. 4(b), the same red dot is represented by two characters *i.e.,* "21". By increasing the number of characters in the hash representation, we reduce the area covered by the hash string. The GeoHash representation is used as the key for Redis datastore. A spatial search is performed for all the locations (*i.e.,* data points) within the set governed by this key. The more characters we have in the GeoHash string, the smaller the search space, thus reducing the area being searched. However, too fine-grained a key could result in no data points being returned by the spatial search, since we search only using the key corresponding to the GeoHash of the queried data point. We explore the implications of the size of GeoHash in the evaluations (§5.1).

3.2 On-Device Bandwidth Manager

Fig. 5 shows the interaction between the on-device components of Foresight, which have been implemented on Android in Java. To enable interaction between the different on-device components, we used inter-process communication (IPC). To specify the communication interfaces, we used Android Interface Definition Language (AIDL) [10]. The bandwidth manager provides three interfaces for client apps to interact with it for bandwidth allocation:

Function Name	Purpose
`registerApp(appName, bandwidthScheduleCallback)`	Provided by the bandwidth manager for a client app to regiser a callback function at startup
`specifyRequirement(appName, bandwidthRequirement)`	Provided by bandwidth manager for a client app to communicate bandwidth requirement
`onBandwidthSchedule(bandwidthInfo)`	Asynchronous notification by the bandwidth manager to the client app using the registered callback to communicate a newly computed bandwidth allocation schedule
`addRoute(route)`	Provided by the bandwidth manager for a route finding application (like maps) to communicate a new route

Table 1: Interface functions for IPC between the bandwidth manager and other on-device components.

(1) `specifyRequirement`: This function allows a client app to communicate its bandwidth requirement to the bandwidth manager. The app can change its requirement midway through its lifetime if it so desires by simply calling the API again. The bandwidth manager then re-computes the allocations in the next epoch and notifies registered applications of the change.
(2) `registerApp`: This function allows a client app to register a callback handler with the bandwidth manager. The callback will be used by the bandwidth manager to asynchronously notify the client app when the bandwidth allocation schedule is ready.
(3) `onBandwidthSchedule`: Implemented by the client app, this callback handler allows the bandwidth manager to supply bandwidth allocation information to the client app asynchronously.

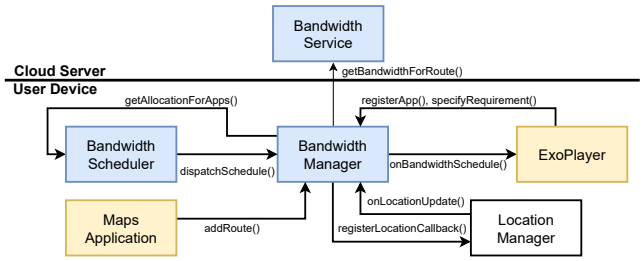

Figure 5: Foresight implementation. The on-device components are implemented as Android apps. Yellow boxes are modified apps to either aid Foresight (Maps), or showcase the utility of Foresight (ExoPlayer). Location Manager is part of the Android Framework.

The bandwidth manager provides an interface that can be called for communicating route information from the user (`addRoute` API as described in Table 1). We built an application that enables the user to share the route information with the bandwidth manager as a series of latitude/longitude pairs. The bandwidth manager also registers a callback with the Android Location Manager to receive asynchronous location updates. Table 1 summarizes the APIs for communication between the components of Foresight. As detailed in §2.2, the bandwidth manager needs several pieces of information to compute the allocation schedule:

User's trajectory: This is obtained from a device resident app such as Maps, via the bandwidth manager's `addRoute` interface.

Spatiotemporal bandwidth information: A background thread periodically queries the on-cloud bandwidth service to receive this information. The periodicity is the *epoch* parameter of Foresight.

App usage statistics: This is obtained from Android's Usage Manager service [13] for the client apps registered with the bandwidth manager. The app usage statistics are used to compute the *value* attribute of the client apps (as described in Section 2).

Estimate of user's speed: Another background thread monitors the user's location for estimating the speed by registering with Android's Location Manager service.

The bandwidth manager maintains a list of applications that have registered callbacks. Once the schedule is computed it is communicated to client apps through the `onBandwidthSchedule` callback corresponding to each app.

4 A Prototype Video Player

To showcase how a streaming application could benefit from Foresight, we have modified the open source video player, *ExoPlayer* [11], such that it adapts its buffer length using the bandwidth allocation schedule provided by Foresight. In this section, we first describe the algorithms we have implemented for adapting the buffer length (§4.1). We then provide the implementation details of how the algorithms are integrated into ExoPlayer in §4.2.

4.1 Buffer Length Adaptation Algorithm

Currently, ExoPlayer [11] relies on historical information for estimating the available bandwidth, which in turn influences the video's bitrate (or quality). While this adaptive bitrate mechanism attempts to absorb small shocks in available bandwidth, it is not designed to compensate for sustained bandwidth drops, *e.g.*, when a user enters a tunnel. In such situations, the video player's buffer drains completely and the player is forced to stall. Under these circumstances the player could benefit from knowing the expected spatiotemporal variations of bandwidth. Prior work has shown that prediction is key to improving video quality of experience, especially in the presence of spatiotemporal variations in bandwidth [20]. Foresight goes a step further; it is "prescient" about upcoming variations through crowd-sourced bandwidth knowledge.

ExoPlayer uses a minimum and maximum buffer length threshold, within which it tries to operate. Bitrate switches happen when either buffer length criteria are not met. ExoPlayer is extensible and allows developers to define their own implementation of various components within the player. We have extended ExoPlayer such that the minimum buffer length can be dynamically varied, based on the future bandwidth estimate that we receive from the scheduler to ensure stall-free operation. It is to be noted that we are not proposing to completely supplant the built-in adaptive bitrate selection algorithms of the player, but to merely augment it with future bandwidth information.

Algorithm 2: Compute Surplus and Deficits

Data: bwInfoByTime, desiredBitrates, confidence
Result: bufferCapacityByTime

```
1  bufferCapacityByTime.surplus = [];
2  bufferCapacityByTime.deficit = [];
3  bitrate_idx = getMinSupportableBitrate(desiredBitratess,
     bwInfoByTime[0].expected_bw);
4  bitrate = desiredBitrates[bitrate_idx];
5  for (i=0; i < bwInfoByTime.size; ++i) do
6      buffer_difference = (bwInfoByTime[i].duration
         *bwInfoByTime[i].expected_bw)/bitrate -
         bwInfoByTime[i].duration;
7      if (buffer_difference > 0) then
8          bufferCapacityByTime.surplus[i] = buffer_difference *
             confidence;
9          bufferCapacityByTime.deficit[i] = 0;
10     else
11         bufferCapacityByTime.deficit[i] =
             abs(buffer_difference));
12         bufferCapacityByTime.surplus[i] =0 ;
13     end
14     bitrate_idx = getMinSupportableBitrate(desiredBitrates,
         bwInfoByTime[i].expected_bw);
15     bitrate = desiredBitrates[bitrate_idx];
16 end
```

Algorithm 2 describes the logic for deciding when the buffer length can be expected to decrease, and when there is excess bandwidth that can be used towards increasing the buffer length. Its inputs are the available bandwidth over time (*bwInfoByTime*), the allowable bitrates in which the video has to be played, and the confidence in the data. *buffer_difference*, in seconds, is computed as the excess download capacity available at each instant, representing the amount of additional data that can be downloaded (measured in seconds of playback time) while accounting for the buffer drain due to playback during that time at the current bitrate. We initially start out with the highest bitrate and switch down or up depending on the expected bandwidth availability (line 3). If we find that even the poorest acceptable quality is unsustainable under the network conditions, we mark the entire duration of low bandwidth availability period as deficit time. While computing the periods of surplus, we apply a confidence factor to account for errors in the bandwidth availability data. Once the surpluses are computed, it is easy to find how much future deficit we need to balance by walking backwards in time and increasing the buffer lengths accordingly.

In this proof of concept, we adopt a simple greedy scheme where we first identify the stall points, and then work backwards in time, increasing the buffer length, contingent on the available surplus, and maximum buffer size. Algorithm 3 shows how we compute the buffer length adjustments. Once we compute the surpluses and deficits, we then identify consecutive periods of deficits (from *deficit_start* to *deficit_end*) and use the preceding surplus to increase the buffer length (lines 3–6). Thus, a portion of the surplus would be used towards compensating for some deficit. The remaining surplus capacity is stored in `updated_surplus`. We then use the difference between the original and updated surplus capacities to determine the increment to the buffer length at each instant.

Algorithm 3: Balance Deficits

Data: deficit_start, deficit_end, bufferCapacity.surplus, bufferCapacity.deficits
Result: updated_surplus

```
1  total_deficit = sum(deficit[deficit_start:deficit_end]);
2  updated_surplus = surplus;
3  for (i =deficit_start -1, i >0 , - -i) do
4      if updated_surplus[i] > 0 then
5          decrement = min(updated_surplus[i], total_deficit);
6          updated_surplus[i] -= decrement;
7          total_deficit -= decrement;
8      end
9  end
10 return updated_surplus;
```

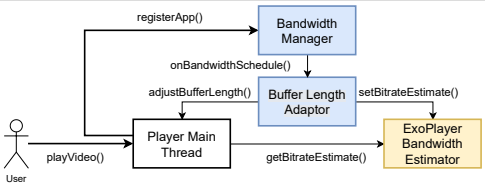

Figure 6: Modifications to the ExoPlayer. The newly added Buffer Length Adaptor module periodically updates ExoPlayer Main Thread, and Bandwidth Estimator based on the computations in Algorithms 2 and 3.

4.2 Implementation Details

Fig. 6 illustrates the modifications we made to the player. In the main thread of the video player application, an instance of ExoPlayer is created. The video player subscribes to updates from the bandwidth manager using AIDL. The *Buffer Length Adaptor* module in Fig. 6 houses the `onBandwidthSchedule` callback function for receiving notification from the bandwidth manager. The buffer adaptation algorithms presented in §4.1 run within this module. Once the buffer length modifications are computed by the Buffer Length Adaptor, they are communicated to the video player and the bandwidth estimator. We modified the bandwidth estimator to directly use the information obtained from the buffer length adaptor module, thereby adding future knowledge into the estimator. The expected result of this modification is to enable ExoPlayer to download enough content ahead of time so that when it goes through periods of low bandwidth availability, it will not have to switch to a lower video bitrate but can afford to simply drain its buffer. We validate this expectation through evaluation of the modified player in §5.3.

5 Performance Evaluations

In this section, we evaluate the performance of the components of Foresight and the modified video player. The evaluations cover (a) the scalability of the on-cloud bandwidth service in handling spatiotemporal bandwidth queries (§5.1), (b) the performance of the on-device bandwidth manager for generating the allocation

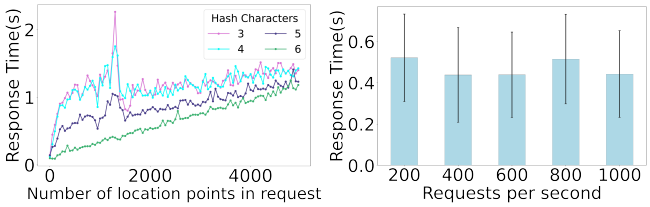

Figure 7: Redis datastore performance. (a) As we increase the number of characters in the GeoHash, the number of points corresponding to the same hash decreases, leading to improved parallelism since there is less contention for the same key. (b) Increasing the number of requests per second does not significantly increase the server response time.

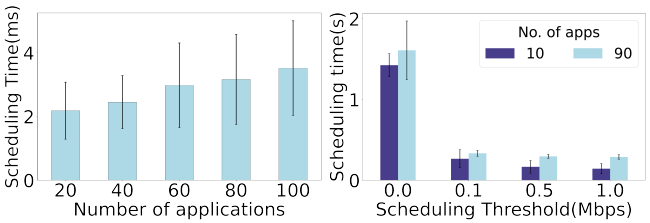

Figure 8: Bandwidth manager performance. (a) As we increase the number of applications scheduling time increases slightly. (b) Scheduling time for 196 bandwidth values. Increasing the change threshold for re-scheduling decreases the time spent in scheduling.

schedules for the clients apps (§5.2), and (c) the ability of the prototype video player to avoid stalls and increase the quality of user experience by utilizing the prescient allocation schedules (§5.3).

5.1 On-Cloud Bandwidth Service Performance

In this section, we evaluate the performance of the spatiotemporal datastore as the load on the cloud server increases. We inspect server performance over two aspects—increased number of query locations per-request, and increased requests per-second.

5.1.1 Experimental Setup

We created a Jetty [8] service which hosts the application threadpool for accessing the Redis [1] spatial datastore and deployed it on an AWS Linux micro instance. The service takes as input a list of latitude/longitude pairs, and returns the expected bandwidth for those locations. For a specified point, we search the datastore for bandwidth readings reported in the vicinity and return the 50 nearest measurements.

5.1.2 Scaling the Number of Locations per Request

When mobile devices query the cloud server to obtain bandwidth estimates for a set of locations, the density of these locations has a direct consequence on the retrieval time and impacts the design decisions concerning location-bandwidth mapping. When storing the computed bandwidth estimates, we varied the number of characters in the GeoHash which is used as the key while querying the spatiotemporal datastore. As mentioned in §3.1.2, the number of characters in the GeoHash, n, determines how much geographical area is mapped to a single key in the Redis datastore. Fig. 7(a) shows the Redis datastore's response time as we increase the number of locations per request. For a smaller value of n, a larger geographical area is mapped to the same key, which results in poor scaling as we increase the number of locations per request. We observe better scalability with larger values of n with close to 1-second response time even for 5000 locations per request. Therefore, for the rest of the evaluations we use a GeoHash size of six characters (*i.e.*, $n = 6$).

5.1.3 Scaling the Number of Requests per Second

The scalability of the bandwidth service is key to the utility of Foresight. Writes to the service occur when crowd-sourced devices run the CRUSP bandwidth estimates and upload their results. We expect this write traffic to be significantly less than the read traffic from mobile clients requesting bandwidth availability for their routes. Therefore, this experiment is focused on read scalability of the datastore. A single instance of the datastore is populated with bandwidth measurements from Ookla's open source dataset [25] for NYC. Using the NYC bus dataset [34] as the workload, the datastore is queried for bandwidth availability on routes. The dataset generates varying number of locations queried per request. The average number of locations per query is 1176 points. There is opportunity for intra-request parallelism (the distinct set of locations queried by each request), and inter-request parallelism (the number of concurrent requests) in handling this workload. Fig. 7(b) shows the per-request response time for requests of variable sizes, as a function of the number of concurrent requests. We observe that the response time is fairly similar as the number of requests increases showing the scalability of the bandwidth service.

5.2 On-Device Bandwidth Manager Performance

This section examines the performance of the bandwidth manager on the device that caters to the concurrent bandwidth allocation needs of the client apps on this device. We evaluate the following aspects: (1) time taken by the bandwidth scheduler to run the knapsack algorithm to allocate bandwidth as a function of time to the concurrent requests; (2) sensitivity of the scheduler to spatial bandwidth variations along the route; and (3) efficacy of adaptive epochs to deal with per-location temporal variations in bandwidth.

5.2.1 Experimental Setup

We implemented the Knapsack algorithm in Java, on an Ubuntu 18.04 Linux machine with 8 GB RAM. We relied on application usage data from an open source Android App usage dataset [24] to derive the "value" parameter for the apps used in the Knapsack algorithm (see §2.2.1).[2] Since the actual bandwidth requirement does not matter in the scheduling time calculations, we simply pick per-application bandwidth requirements from a normal distribution.

5.2.2 Scheduler Scaling with Number of Applications

For each client app that registers with the bandwidth manager and presents a bandwidth requirement for a route, estimates are fetched from the on-cloud bandwidth service for every location in the newly added route. Subsequently, the scheduling algorithm is run for each location in the route for all the registered applications. We find that on an average, the scheduler takes 3 ms to run. As shown in Fig. 8(a), when we increase the number of applications, the time taken by the scheduler increases only by tens of microseconds confirming the scalability of the scheduler.

[2]The bandwidth allocation on a device depends on an individual user's application usage patterns. Therefore, in the actual implementation detailed in §3, we derive the "value" parameter based on the individual's usage, based on statistics collected from Android's Usage Manager service [13].

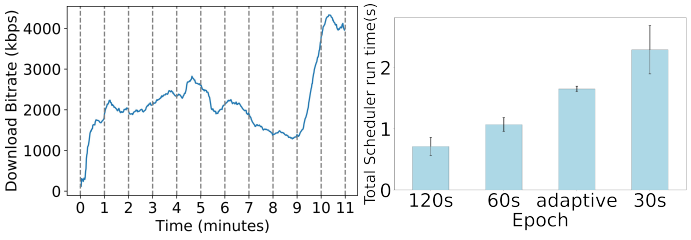

Figure 9: (a) Exemplar temporal variation in bandwidth for a single location from Cork dataset. (b) Total scheduling time for 90 apps for various epoch sizes.

5.2.3 Scheduler Sensitivity to Spatial Bandwidth Variations

In this section, we investigate the effect of spatial variability in bandwidth on scheduling. Often there are small changes in the available bandwidth between adjacent locations of a route. To bound the number of times the Knapsack algorithm is run to generate the allocation schedule, we introduced the concept of *change threshold* (see §2.2.1). Fig. 8(b) captures the differences in the running time of the scheduler over the entire route as we increase the *change threshold* from 0 (always recompute), to 1 Mbps. We observe that setting a 1 Mbps *change threshold* produces a saving of roughly 1 second when scheduling for a route with 196 location points. Increasing the number of applications subscribing to the scheduler also sees a slight (but not significant) increase in the scheduling time. Choosing the right threshold would depend on the bandwidth estimates received. In the route that we used, the mean difference in bandwidth between adjacent locations is 2.46 Mbps. So, setting even a 500 Kbps threshold will not reduce the accuracy of the scheduling by a large amount.

5.2.4 Evaluating Adaptive Epochs

Fig. 9(a) illustrates the variation in bandwidth for a single point (*i.e.,* location) in a dataset that contains real life network measurements for several locations in Cork City [30]. The bandwidth remains largely stable between 1 and 9 minutes, after which there is a huge spike. It is precisely for this reason that we introduced *adaptive epochs* in §2.2.2, such that we query the cloud server more often when temporal variations in bandwidth availability are observed. Under stable bandwidth conditions, querying the cloud server would unnecessarily waste network resources, and compute resources on the cloud and the device.

To study the utility of adaptive epochs, we first we created a base bandwidth availability trace and perturbed it with temporal variations at specific points along the trace and obtained the bandwidth information that would be seen by the fixed and adaptive epochs. The adaptive epoch started with a base of 20 seconds. We incremented the epoch size by 10 seconds if no changes to bandwidth measurements were observed. If there were changes, we reduced the epoch size to half its current value. The fixed epoch is set to 60 seconds. The result of this experiment is shown in Fig. 10. The pink line shows the base bandwidth availability trace. The thick light green line depicts the actual bandwidth with the temporal variations we introduced through the perturbations. The cyan line depicts the behavior of the fixed 60 second epoch setting, which is able to pick up only some temporal variations (*e.g.,* the bandwidth changes for the portion of the route corresponding to 150–200 seconds) but misses others; on the other hand, the adaptive epoch (dark blue line) is able to see all the variations in bandwidth (including the one between 90 and 120 seconds missed by the periodic epoch).

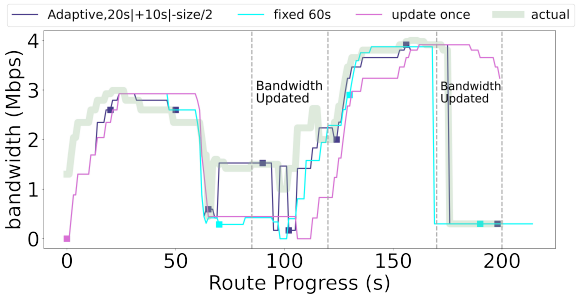

Figure 10: Efficacy of adaptive epoch. Temporal bandwidth variations are closely tracked by the adaptive epoch policy.

Next, we study the effect of the epoch size on the scheduling overhead incurred on the device. The experiment uses 90 client apps each with a 5 minute travel duration. As can be seen in Fig. 9(b), the larger the epoch size the smaller the scheduling overhead. The adaptive epoch, with an initial size of 20 seconds represents a compromise in terms of scheduling overhead between the two periodic epochs of 60 seconds and 30 seconds.

5.3 Prototype Video Player Performance

In this section, we evaluate the benefits of prescient bandwidth availability provided by Foresight in a real application, *i.e.,* ExoPlayer. Our intent is to show that a streaming application could use this information to optimize specific parameters which would ultimately improve the user's quality of experience (QoE). Specifically, using ExoPlayer as the example, we show that this information helps to improve the QoE by (1) providing a higher-bitrate playback whenever possible, or (2) offering playback with fewer bitrate switches, or (3) reducing the number of stalls experienced during playback. The specific metric to optimize for, is in the app's control.

As mentioned in §4.2, we have incorporated our buffer length adaptation modifications into ExoPlayer with the intent to influence the player's built-in adaptive bitrate (ABR) selection algorithm. Our intuition is that since the video player switches from a higher bitrate to a lower bitrate when it is unable to maintain a certain threshold buffer length, by manipulating the buffer-lengths, the bitrate switching behavior can be altered.

The controlled experiments in this section showcase the utility of Foresight for ExoPlayer along the following dimensions: (a) adaptation of video QoE to spatial bandwidth variations (§5.3.2); (b) adaptation of video QoE to temporal bandwidth variations (§5.3.3); and (c) overall performance on different bandwidth profiles (§5.3.4).

5.3.1 Experimental Setup

For experiments on the video player, we used the following setup. The video segments in DASH compatible format were obtained from BBC's open source repository [3] and hosted on Apache2 server running on Ubuntu 20.04. This server provides the video segments to the video player which runs on a OnePlus 6A Android phone. The video segment server and the phone are on the same local area network.

Emulating spatiotemporal bandwidth variations. In order to emulate the spatial and temporal variations in bandwidth, we used

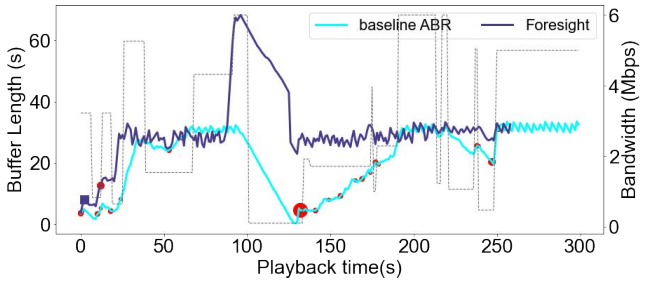

Figure 11: Video QoE for spatial bandwidth variations. Comparison of Exoplayer's baseline ABR and Modified ExoPlayer that uses Foresight.

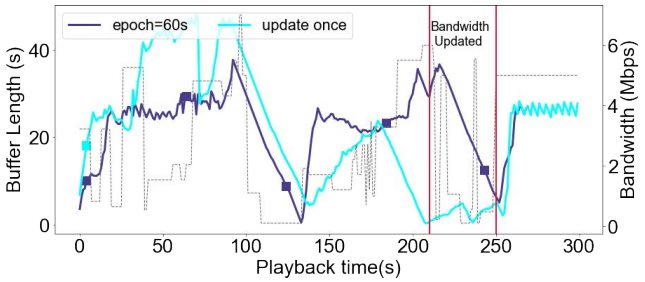

Figure 12: Video QoE for temporal bandwidth variations. Comparison of different epoch sizes on timely notification to the video player. Cyan line is for epoch size of infinity (*i.e.,* notify once); Dark blue line is for epoch size of 60 secs. Square dots denote notification from the scheduler. The gray line is the available bandwidth profile. Between the red lines, the available bandwidth changed from the initial estimates to a low of 0.3 Mbps.

bandwidth measurements from the Cork City dataset. To emulate a real-life network experience on the video player, we first use Google Maps API [12] to get latitude/ longitude values along a chosen route, and then get the bandwidth measurements for those points from the Cork City dataset. Using these bandwidth values, we then perform egress traffic shaping on the video segment server using Linux Traffic Control (*tc*). We gather video player metrics such as buffer length and video bitrates at 2-second intervals.

5.3.2 Video QoE on Spatial Bandwidth Variations

For this experiment, we traffic shape a specific available bandwidth profile (light gray line in Fig. 11) to mimic spatial variations in the bandwidth along a route. To study the effect of network conditions on video quality of experience, we introduce periods of extremely low bandwidth (tens of Kbps) along the route. We consider two situations: (1) We let the vanilla ExoPlayer experience this bandwidth profile and use its baseline built-in ABR algorithm to handle fluctuations in the available bandwidth along the route. (2) We use the on-device bandwidth manager to communicate the allocation schedule to our modified ExoPlayer *once*, commensurate with this bandwidth profile. As shown in Fig. 11, adding the knowledge of expected download bandwidth can enhance the user's quality of experience by helping the modified ExoPlayer avoid stalls and extreme bitrate switches. At the beginning, when the player gets the bandwidth schedule from the scheduler, it adapts the buffer length to insulate against future drops. At about 150 seconds, the baseline player (cyan line) runs out of buffered content when the bandwidth has already dropped to tens of Kbps, resulting in a stall in the baseline player since there is no more buffered video content to play. The player tries to mitigate the situation by reducing the bitrate, but it is unable to do so because there is insufficient bandwidth. The bitrate switches are numerous in the baseline player, as illustrated by the red dots. The larger the bitrate switch (*i.e.,* the video quality changes drastically), the larger the radius of the red dots. In comparison, the modified ExoPlayer (dark blue line) *knows* that the bandwidth is expected to fall in the next one minute, and so, at about 80 seconds on the player timeline, it triggers an increase in buffer length, which causes more segments to be downloaded. Thus, when the bandwidth falls at around 100-second mark, the player is able to ensure stall-free operation since it has already accumulated enough content. We note that while our modified player does experience bitrate switches, they are much less perceptible and fewer in number (2 in comparison to the baseline player's more than 10 switches).

5.3.3 Video QoE on Temporal Bandwidth Variations

This experiment is designed to study the effect of timely updates of temporal variations in bandwidth available to the modified ExoPlayer. At the beginning of every epoch, Foresight fetches fresh bandwidth estimates and applies them to the allocation schedules of client apps.

To investigate the effect of changing the epoch size on ExoPlayer's behavior, we added small variations to the egress bandwidth in the experimental setup introduced in §5.3.1. We introduced a new low bandwidth period starting at 211 seconds, up to 250 seconds. Then, we added logic in the scheduler such that if the location/bandwidth information is queried 1 minute after the start of the experiment, then the new low bandwidth values would be made available. We studied the effect of epoch size on the performance of the video player. In Fig. 12, when the bandwidth estimates are updated periodically (every 60 seconds) by the scheduler (dark blue line), the player is able to suitably update its buffer and handle the low bandwidth region at 211 to 250 seconds. The player completely avoids stalling, whereas, if the bandwidth estimates are only received at the beginning and not updated afterwards (cyan line), the player starts stalling since it did not plan for the low bandwidth region.

5.3.4 Robustness to Various Bandwidth Profiles

This experiment is aimed at studying the response of the modified ExoPlayer to different bandwidth profiles. The purpose of this experiment is to understand the utility of Foresight in catering to any chosen QoE metric of importance to the player. For this experiment, we have chosen to optimize number of bitrate switches.

To remove bias in the evaluation, we use the methodology outlined in §5.3.1 to create bandwidth profiles for randomly generated source/destination pairs in the Cork Dataset. From these bandwidth profiles, we then select 14 routes with at least one period of low bandwidth (<500Kbps bandwidth along the route). For each of these routes, we perform traffic shaping on the video segment server commensurate with these bandwidth profiles. We assume no temporal variations for this experiment and thus it is sufficient if these profiles are fed to the modified ExoPlayer *once* at the beginning of the journey (*i.e.,* epoch size is infinity). We measure the periods of time

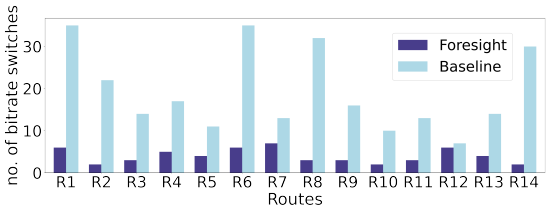

Figure 13: Robustness to varied bandwidth profiles. Foresight-enhanced modified ExoPlayer results in fewer bitrate switches on all routes

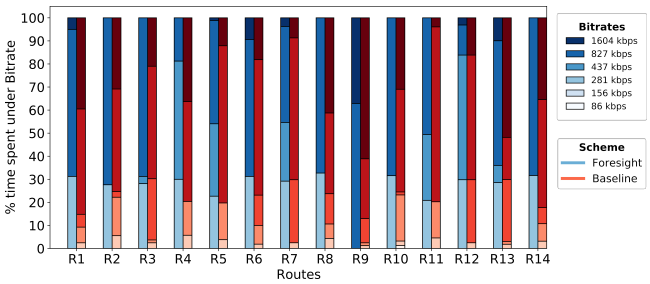

Figure 14: Robustness to varied bandwidth profiles. Darker shading (for both blue and red) implies higher bitrate. Foresight-enhanced modified ExoPlayer results in larger percentage of time in higher bitrates.

for which the video player played the video at various bitrates, for the unmodified ExoPlayer, and Foresight-enhanced ExoPlayer.

Fig. 13 shows the number of bitrate switches on each of the 14 routes. Adding the buffer length adjustment logic can produce a better quality of experience by virtue of fewer bitrate switches. Since the player knows ahead of time the variations in bandwidth for the future, it is able to buffer content at a mostly stable bitrate than the baseline. We observed that for the most part, the adaptive buffer length player is able to avoid playing at the lowest bitrate on most routes, as illustrated in Fig. 14. However, it can be seen that with the addition of buffer length adaptation, the duration of playback at the highest bitrate is reduced in favor of a slightly lower bitrate so that there is enough bandwidth to prefetch segments for avoiding future bitrate switches. We found that stall times for both players were similar. Thus, it is safe to conclude that buffer length adaptation did not come at the expense of increased stalling.

6 Related Work

Crowd-sourced Bandwidth Estimation. Crowd-sourced spatiotemporal bandwidth measurement is employed by companies like speedtest.net [26] by Ookla and OpenSignal [27]. While Ookla has open-sourced some of its data, it is hard to get a reliable live estimate of the experienced bandwidth. Dubin, *et al.* [7] propose using crowd-sourced bandwidth information from a dataset provided by Wefi Ltd to adapt video bitrates for on-demand streaming. While the dataset accounts for spatiotemporal variations in bandwidth, it is difficult to track real time variations in expected bandwidth. Such datasets are however, invaluable for bootstrapping the service in new locations.

Bandwidth Estimation Techniques. In the realm of bandwidth estimation, prior works can be categorized into passive or active methods [2]. Passive methods monitor and analyze the inbound and outbound traffic on a network interface. However, the reliability of these methods is often influenced by user activity.

Active methods involve saturating the network with packets to reliably estimate bandwidth. These methods are employed by OpenSignal or Ookla to carry out bandwidth measurements. Ookla provides users with an application which tries to open multiple TCP connections, and measure the latency and throughput experienced while exchanging large volumes of data with a nearby speedtest.net server. The size of the data is increased slowly until the network becomes congested, thereby providing an estimate of the maximum end-to-end throughput, originating at a given location, for a given wired or wireless connection. However, active probing methods come with the overhead of consuming user data to estimate bandwidth.

Low overhead active probing methods include packet dispersion techniques [29] that estimate the capacity between two nodes by studying the latency between two packets on the bottleneck link connecting the two nodes. Such techniques are often unreliable in cellular networks because packets tend to arrive in bursts [37]. CRUSP [31], however is a low overhead probing tool which is more suitable for mobile landscapes because it uses a burst (2MB in size) of data to estimate link capacity.

Bandwidth Prediction Models. In most prior works [5, 14, 33], the bandwidth prediction model is built on HTTP packet data explicitly collected while streaming videos-on-demand. By contrast, Xu, *et al.*, passively monitor the network by building a library which relies on application traffic to collect network information, using timing and sequence numbers [36]. They then build a regression tree to predict temporal variations in bandwidth. However, it is not clear if user mobility is accounted for in their work. Yue *et al.* [38] quantify the importance of both low level (*e.g.*, channel quality) and high level (*e.g.*, historical link bandwidth) statistics to predict instantaneous bandwidth using a machine learning model. Sprout[35] is an end-to-end transport layer protocol which also forecasts instantaneous bandwidth for use by high throughput applications. Mei *et al.*, instead actively flood the network using iPerf [16] to obtain traces of cellular bandwidth availability under various conditions, and then create a neural network classifier to predict the temporal variations in bandwidth [21].

Using Bandwidth Estimates for Packet Scheduling. A number of prior works utilize knowledge about parameters such as signal strength, predicted bandwidth and physical layer statistics in order to make better packet scheduling decisions. Rathnayake, *et al.*, propose Emune which comprises a bandwidth prediction component and a scheduling engine, exposed via an API for applications to use [32]. Within the scheduling engine, they perform packet scheduling using stochastic optimization methods, with the goal of minimizing power costs and monetary usage. Each application has a priority and certain scheduling constraints like hard and soft deadlines for packet transfer. While this approach seems promising, it is also extremely restricting; we take a more suggestive approach instead, leaving it up to applications to decide how to use the information about bandwidth availability. Further, we make use of crowd-sourced spatiotemporal bandwidth information to inform our scheduler instead of just relying on metrics like physical layer statistics derived from the device itself.

Bandwidth Estimation Baked into Video Players. There are also several works which have proposed the idea of a location/throughput database [5, 33] where the user's route information is shared with a bandwidth service which then aggregates/updates the data and informs the player running on the user's device about the expected bandwidth in the future. G-Tube [14] does not require the entire route information, but instead combines the uploading of location data to the server with the functionality of the video player. While this design serves the purpose of improving video streaming QoE, we opt to decouple the two tasks in order to achieve greater flexibility in deployment and wider utility across different apps.

Bitrate Adaptation in Video Players. There is prior art in improving adaptive bitrate selection usually based on bandwidth estimation and a minimum buffer length criteria [6, 19, 20]. Such proposals suggest criteria for deciding when to switch to a lower bitrate, based on the current state of the buffer or predicted bandwidth. Such techniques have been incorporated into various open source implementations of adaptive bitrate streaming players such as dash.js [9] and ExoPlayer [11], as well well known video players including YouTube and Netflix.

7 Discussion

Bandwidth Estimation. The aggregation mechanism used to estimate available bandwidth for a location is the running mean of bandwidth estimates received within an hour. This scheme assumes that the estimates are independent of the number of crowd-sourced users reporting them for the same location. The physical layer communication technology used by the mobile devices could have an impact on the validity of this assumption. There is room for refining the aggregation mechanism to account for overlap in space and time of crowd-sourced data to improve the accuracy of the estimates recorded by the bandwidth service in the Cloud. While this work is evaluated for a single-carrier's bandwidth estimates, Foresight can be personalized for multiple carriers in the future.

Scheduling Accuracy. Beyond the accuracy of the bandwidth estimates used by the scheduler, the epoch granularity has a significant impact on the accuracy of the schedules. If the temporal variations in bandwidth have a finer granularity than the epoch duration, it is possible for the scheduling to be sub-optimal. At the other extreme, we could have no estimates for some regions due to unavailability of crowd-sourced data. One way to mitigate the data sparsity problem is to use a larger area for the spatial search. Another approach is to augment the crowd-sourced data with publicly available data sets. Further, to account for scheduling inaccuracies, client apps could dynamically vary the confidence factor in using the schedules.

Usage Statistics. We have relied on the app usage data from Android for deriving the "value" parameter in solving the Knapsack problem that makes allocation decisions for the client apps. The limitation of this approach is that the statistics are available only in two hour buckets, so finer variations in app usage patterns will not be captured by the scheduler. One approach to solving this problem is for the bandwidth manager to track app usage statistics on its own over time.

Scalability of the On-cloud Bandwidth Service. The current implementation uses a single Redis instance. As the size of the data grows over time, we would need to consider partitioning the bandwidth estimation data on a cluster of Redis nodes to increase the parallelism for serving device requests. In densely populated areas, we can expect location/bandwidth estimates to be more fine grained than sparsely populated regions. Therefore, the number of characters used in the GeoHash keys, both for the creation and subsequent retrieval of bandwidth information, should be chosen keeping this disparity in mind.

Utility of Foresight in a Video Player. We added logic to dynamically change the buffer length in a video player, when the bandwidth is expected to fall in the near future, to avoid fetching segments at a lower bitrate. The performance of this scheme hinges on two key factors: (1) currently experienced bandwidth, and (2) duration of the low bandwidth region. If the current bandwidth is only marginally better than the future bandwidth estimate, there is only a limited extent to which adjusting buffer length can help, since the player might not have enough surplus bandwidth to fetch extra segments. Further, if the duration of low bandwidth far exceeds the "good" bandwidth region, then a stall can only be postponed, but not completely avoided. In such situations, at best the player can reduce the bitrate to avoid stalling longer.

Additional Work in Client Apps for using Foresight. A client app wishing to use the results from Foresight would need some code modifications. We found the modification to be straightforward and minimal in ExoPlayer (230 additional lines of code). Further, the measured additional overhead in ExoPlayer for calculating the buffer adjustment schedule is less than 35 ms per run. We have implemented this activity in a background thread without affecting the main playback. It should also be noted that this overhead decreases with time of journey of the mobile user as the playback time for the video continues to get shorter.

8 Conclusion

Foresight is an end-to-end system to cater to the needs of users on the move who expect a high quality of experience for streaming apps running on their mobile devices. It addresses the problem of spatiotemporal variations in bandwidth availability by incorporating three components in its design, *i.e.,* a crowd-sourced bandwidth estimation facility, an on-cloud bandwidth service for recording the estimates and serving mobile users, and an on-device bandwidth manager to allocate the available bandwidth to the client apps that need them. Through location-obfuscation, Foresight also obviates the need for client apps to request permission for location knowledge from the mobile device. To showcase the utility of Foresight, we have modified ExoPlayer, an open-source video player to incorporate the results of Foresight in its video buffer management to enhance the QoE for mobile users.

9 Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback, which substantially improved the content and presentation of this paper. This work was funded in part by NSF I/UCRC FiWIN Center (IIP-1821819), NSF CNS-1909346, and a gift from Microsoft Corp.

References

[1] 2020. Redis. https://redis.io/

[2] T. Arsan. 2012. Review of bandwidth estimation tools and application to bandwidth adaptive video streaming. In *High Capacity Optical Networks and Emerging/Enabling Technologies*. 152–156. https://doi.org/10.1109/HONET.2012.6421453

[3] BBC. 2020. Exoplayer Testing Samples. https://github.com/bbc/exoplayer-testing-samples

[4] United States Census Bureau. 2020. TIGER/Line Shapefiles. https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html

[5] Igor D.D. Curcio, Vinod Kumar Malamal Vadakital, and Miska M. Hannuksela. 2010. Geo-Predictive Real-Time Media Delivery in Mobile Environment. In *Proceedings of the 3rd Workshop on Mobile Video Delivery* (Firenze, Italy) *(MoViD '10)*. Association for Computing Machinery, New York, NY, USA, 3âĂŞ8. https://doi.org/10.1145/1878022.1878036

[6] Dongeun Suh, Insun Jang, and Sangheon Pack. 2014. QoE-enhanced adaptation algorithm over DASH for multimedia streaming. In *The International Conference on Information Networking 2014 (ICOIN2014)*. 497–501. https://doi.org/10.1109/ICOIN.2014.6799731

[7] Ran Dubin, Amit Dvir, Ofir Pele, Ofer Hadar, Itay Katz, and Ori Mashiach. 2018. Adaptation logic for HTTP dynamic adaptive streaming using geo-predictive crowdsourcing for mobile users. *Multimedia Systems* 24, 1 (2018), 19–31.

[8] Eclipse. 2020. Eclipse Jetty. https://www.eclipse.org/jetty/

[9] DASH Industry Forum. 2020. dash.js. https://github.com/Dash-Industry-Forum/dash.js/wiki/Low-Latency-streaming

[10] Google. 2020. AIDL. https://developer.android.com/guide/components/aidl#Expose

[11] Google. 2020. ExoPlayer. https://exoplayer.dev/

[12] Google. 2020. Google Maps Developer API. https://developers.google.com/maps/documentation

[13] Google. 2020. UsageStatsManager. https://developer.android.com/reference/android/app/usage/UsageStatsManager

[14] Jia Hao, Roger Zimmermann, and Haiyang Ma. 2014. GTube: Geo-Predictive Video Streaming over HTTP in Mobile Environments. In *Proceedings of the 5th ACM Multimedia Systems Conference* (Singapore, Singapore) *(MMSys '14)*. Association for Computing Machinery, New York, NY, USA, 259âĂŞ270. https://doi.org/10.1145/2557642.2557647

[15] Wolfgang Hofer. 2020. Infrastructure for Mobile. https://gitlab.com/gitlabwolf/infrastructure-for-mobile

[16] iPerf. 2020. iPerf. https://iperf.fr/

[17] U. Karabulut, A. Awada, I. Viering, M. Simsek, and G. P. Fettweis. 2018. Spatial and Temporal Channel Characteristics of 5G 3D Channel Model with Beamforming for User Mobility Investigations. *IEEE Communications Magazine* 56, 12 (2018), 38–45. https://doi.org/10.1109/MCOM.2018.1800218

[18] Lettuce. 2020. Lettuce. https://lettuce.io/

[19] Chenghao Liu, Imed Bouazizi, and Moncef Gabbouj. 2011. Rate adaptation for adaptive HTTP streaming. In *Proceedings of the second annual ACM conference on Multimedia systems*. 169–174.

[20] Tarun Mangla, Nawanol Theera-Ampornpunt, Mostafa Ammar, Ellen Zegura, and Saurabh Bagchi. 2016. Video through a crystal ball: Effect of bandwidth prediction quality on adaptive streaming in mobile environments. In *Proceedings of the 8th International Workshop on Mobile Video*. 1–6.

[21] Lifan Mei, Runchen Hu, Houwei Cao, Yong Liu, Zifa Han, Feng Li, and Jin Li. 2019. Realtime mobile bandwidth prediction using lstm neural network. In *International Conference on Passive and Active Network Measurement*. Springer, 34–47.

[22] Netflix. 2020. Fast. https://www.fast.com

[23] Gustavo Niemeyer. 2008. Geohash. *Retrieved June* 6 (2008), 2018.

[24] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (Roma, Italy) *(SenSys '13)*. Association for Computing Machinery, New York, NY, USA, Article 10, 14 pages. https://doi.org/10.1145/2517351.2517354

[25] Ookla. 2020. Ookla Open Data. https://github.com/teamookla/ookla-open-data

[26] Ookla. 2020. SpeedTest Servers. https://www.ookla.com/speedtest-servers

[27] OpenSignal. 2020. Methodology Overview. https://www.opensignal.com/methodology-overview

[28] Eoin OâĂŹConnell, Denis Moore, and Thomas Newe. 2020. Challenges Associated with Implementing 5G in Manufacturing. In *Telecom*, Vol. 1. Multidisciplinary Digital Publishing Institute, 48–67.

[29] Vern Paxson. 1997. End-to-end Internet packet dynamics. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*. 139–152.

[30] Darijo Raca, Jason J. Quinlan, Ahmed H. Zahran, and Cormac J. Sreenan. 2018. Beyond Throughput: A 4G LTE Dataset with Channel and Context Metrics. In *Proceedings of the 9th ACM Multimedia Systems Conference* (Amsterdam, Netherlands) *(MMSys '18)*. Association for Computing Machinery, New York, NY, USA, 460âĂŞ465. https://doi.org/10.1145/3204949.3208123

[31] V. Raida, P. Svoboda, M. Kruschke, and M. Rupp. 2019. Constant Rate Ultra Short Probing (CRUSP): Measurements in Live LTE Networks. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 1–6. https://doi.org/10.1109/ICC.2019.8761179

[32] Upendra Rathnayake, Henrik Petander, Maximilian Ott, and Aruna Seneviratne. 2012. Emune: Architecture for mobile data transfer scheduling with network availability predictions. *Mobile Networks and Applications* 17, 2 (2012), 216–233.

[33] Haakon Riiser, Tore Endestad, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. 2012. Video Streaming Using a Location-Based Bandwidth-Lookup Service for Bitrate Planning. *ACM Trans. Multimedia Comput. Commun. Appl.* 8, 3, Article 24 (Aug. 2012), 19 pages. https://doi.org/10.1145/2240136.2240137

[34] Michael Stone. 2020. New York City Bus Data. https://www.kaggle.com/stoney71/new-york-city-transport-statistics

[35] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 459âĂŞ472.

[36] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. 2013. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (Taipei, Taiwan) *(MobiSys '13)*. Association for Computing Machinery, New York, NY, USA, 347âĂŞ360. https://doi.org/10.1145/2462456.2464453

[37] Yin Xu, Zixiao Wang, Wai Kay Leong, and Ben Leong. 2014. An End-to-End Measurement Study of Modern Cellular Data Networks. In *Passive and Active Measurement*, Michalis Faloutsos and Aleksandar Kuzmanovic (Eds.). Springer International Publishing, Cham, 34–45.

[38] Chaoqun Yue, Ruofan Jin, Kyoungwon Suh, Yanyuan Qin, Bing Wang, and Wei Wei. 2018. LinkForecast: Cellular Link Bandwidth Prediction in LTE Networks. *IEEE Transactions on Mobile Computing* 17, 7 (2018), 1582–1594. https://doi.org/10.1109/TMC.2017.2756937