Fast and Light Bandwidth Testing for Internet Users

Xinlei Yang[1*], Xianlong Wang[1*], Zhenhua Li[1], Yunhao Liu[1]
Feng Qian[2], Liangyi Gong[1], Rui Miao[3], Tianyin Xu[4]
[1]*Tsinghua University* [2]*University of Minnesota* [3]*Alibaba Group* [4]*UIUC*

Abstract

Bandwidth testing measures the access bandwidth of end hosts, which is crucial to emerging Internet applications for network-aware content delivery. However, today's bandwidth testing services (BTSes) are slow and costly—the tests take a long time to run, consume excessive data usage at the client side, and/or require large-scale test server deployments. The inefficiency and high cost of BTSes root in their methodologies that use excessive temporal and spatial redundancies for combating noises in Internet measurement.

This paper presents FastBTS to make BTS fast and cheap while maintaining high accuracy. The key idea of FastBTS is to accommodate and exploit the noise rather than repetitively and exhaustively suppress the impact of noise. This is achieved by a novel statistical sampling framework (termed *fuzzy rejection sampling*). We build FastBTS as an end-to-end BTS that implements fuzzy rejection sampling based on elastic bandwidth probing and denoised sampling from high-fidelity windows, together with server selection and multi-homing support. Our evaluation shows that with only 30 test servers, FastBTS achieves the same level of accuracy compared to the state-of-the-art BTS (`SpeedTest.net`) that deploys ∼12,000 servers. Most importantly, FastBTS makes bandwidth tests 5.6× faster and 10.7× more data-efficient.

1 Introduction

Access link bandwidth of Internet users commonly constitutes the bottleneck of Internet content delivery, especially for emerging applications like AR/VR. In traditional residential broadband networks, the access bandwidth is largely stable and matches ISPs' service plans [9, 14, 15]. In recent years, however, it becomes less transparent and more dynamic, driven by virtual network operators (VNOs), user mobility, and infrastructure dynamics [21].

To effectively measure the access bandwidth, bandwidth testing services (BTSes) have been widely developed and deployed. BTSes serve as a core component of many applications that conduct network-aware content delivery [1, 31]. BTSes' data are cited in government reports, trade press [37], and ISPs' advertisements [29]; they play a key role in ISP customers' decision making [39]. During COVID-19, BTSes are top "home networking tips" to support telework [11, 12]. The following lists a few common use cases of BTSes:

- VNO has been a popular operation model that resells network services from base carrier(s). The shared nature of VNOs and their complex interactions with the base carriers make it challenging to ensure service qualities [69, 72, 78]. Many ISPs and VNOs today either build their own BTSes [2], or recommend end users to use public BTSes. For example, `SpeedTest.net`, a popular BTS, serves more than 500M unique visitors per year [4].
- Wireless access is becoming ubiquitous, exhibiting heterogeneous and dynamic performance. To assist users to locate good coverage areas, cellular carriers offer "performance maps" [16], and several commercial products (*e.g.,* WiFiMaster used by 800M mobile devices [31]) employ crowd-sourced measurements to probe bandwidth.
- Emerging bandwidth-hungry apps (*e.g.,* UHD videos and VR/AR), together with bandwidth-fluctuating access networks (*e.g.,* 5G), make BTSes an integral component of modern mobile platforms. For example, the newly released Android 11 provides 5G apps with a bandwidth estimation API that offers "a rough guide of the expected peak bandwidth for the first hop of the given transport [20]."

Most of today's BTSes work in three steps: (1) setup, (2) bandwidth probing, and (3) bandwidth estimation. During the setup process, the user client measures its latency to a number of candidate test servers and selects one or more servers with low latency. Then, it probes the available bandwidth by uploading and downloading large files to and from the test server(s) and records the measured throughput as samples. Finally, it estimates the overall downlink/uplink bandwidth.

The key challenge of BTSes is to deal with *noises* of Internet measurements incurred by congestion control, link sharing, *etc.* Spatially, the noise inflates as the distance (the routing hop count) increases between the user client and test server. Temporally, the throughput samples may be constantly fluctuating over time—the shorter the test duration is, the severer impact on throughput samples the noise can induce. An effective BTS needs to accurately and efficiently measure the access bandwidth from noisy throughput samples.

Today's BTSes are slow and costly. For example, a 5G bandwidth test using `SpeedTest.net` for a 1.15 Gbps downlink takes 15 seconds of time and incurs 1.94 GB of data usage on end users in order to achieve satisfying test accuracy. To deploy an effective BTS, hundreds to thousands of test servers are typically needed. Such a level of cost (both at the

* Co-primary authors. Zhenhua Li is the corresponding author.

client and server sides) and long test duration prevent BTSes from being a foundational, ubiquitous Internet service for high-speed, metered networks. Based on our measurements and reverse engineering of 20 commercial BTSes (§2), we find that the inefficiency and cost of these BTSes fundamentally root in their methodology of relying on temporal and/or spatial redundancy to deal with noises:

- Temporally, most BTSes rely on a *flooding-based* bandwidth probing approach, which simply injects an excessive number of packets to ensure that the bottleneck link is saturated by test data rather than noise data. Also, their test processes often intentionally last for a long time to ensure the convergence of the probing algorithm.
- Spatially, many BTSes deploy dense, redundant test servers close to the probing client, in order to avoid "long-distance" noises. For example, FAST.com and SpeedTest.net deploy ∼1,000 and ∼12,000 geo-distributed servers, respectively, while WiFiMaster controversially exploits a large Internet content provider's CDN server pool.

In this paper, we present FastBTS to make BTS fast and cheap while maintaining high accuracy. Our key idea is to accommodate and exploit the noise through a novel statistical sampling framework, which eliminates the need for long test duration and exhaustive resource usage for suppressing the impact of noise. Our insight is that the workflow of BTS can be modeled as a process of *acceptance-rejection sampling* [43] (or *rejection sampling* for short). During a test, a sequence of throughput samples are generated by bandwidth probing and exhibit a measured distribution $P(x)$, where x denotes the throughput value of a sample. They are filtered by the bandwidth estimation algorithm, in the form of an acceptance-rejection function (ARF), which retains the accepted samples and discards the rejected samples to model the target distribution $T(x)$ for calculating the final test result.

The key challenge of FastBTS is that $T(x)$ cannot be known beforehand. Hence, we cannot apply traditional rejection sampling algorithm that assumes a $T(x)$ and uses it as an input. In practice, our extensive measurement results show that, while the noise samples are scattered across a wide throughput interval, the true samples tend to concentrate within a narrow throughput interval (termed as a *crucial interval*). Therefore, one can reasonably model $T(x)$ using the crucial interval, as long as $T(x)$ is persistently covered by $P(x)$. We name the above-described technique *fuzzy rejection sampling*.

FastBTS implements fuzzy rejection sampling with the architecture shown in Figure 1. First, it narrows down $P(x)$ as the boundary of $T(x)$ to bootstrap $T(x)$ modeling. This is done by an Elastic Bandwidth Probing (EBP) mechanism to tune the transport-layer data probing rate based on its deviation from the currently-estimated bandwidth. Second, we design a Crucial Interval Sampling (CIS) algorithm, acting as the ARF, to efficiently calculate the optimal crucial interval with throughput samples (*i.e.,* performing denoised sampling from high-fidelity throughput windows). Also, the Data-driven Server Selection (DSS) and Adaptive Multi-Homing (AMH) mechanisms are used to establish multiple parallel connections with different test servers when necessary. DSS and AMH together can help saturate the access link, so that $T(x)$ can be accurately modeled in a short time, even when the access bandwidth exceeds the capability of each test server.

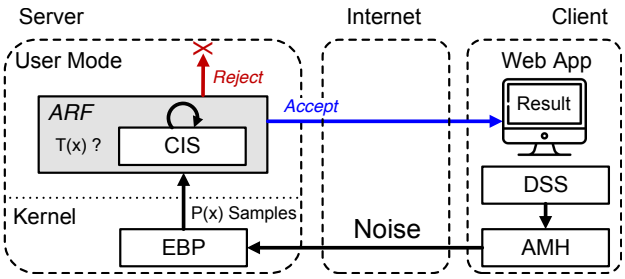


Figure 1: An architectural overview of FastBTS. The arrows show the workflows of a bandwidth test in FastBTS.

We have built FastBTS as an end-to-end BTS, consisting of the FastBTS app for clients, and a Linux kernel module for test servers. We deploy the FastBTS backend using 30 geo-distributed budget servers, and the FastBTS app on 100+ diverse client hosts. Our key evaluation results are[1]:

- On the same testbed, FastBTS yields 5%–72% higher average accuracy than the other BTSes under diverse network scenarios (including 5G), while incurring 2.3–8.5× shorter test duration and 3.7–14.2× less data usage.
- Employing only 30 test servers, FastBTS achieves comparable accuracy compared with the production system of SpeedTest.net with ∼12,000 test servers, while incurring 5.6× shorter test duration and 10.7× less data usage.
- FastBTS flows incur little (<6%) interference to concurrent non-BTS flows—EBP only ramps up fast when the data rate is well below the available bandwidth; it slowly grows the data rate when it is about to hit the bottleneck bandwidth.

To benefit the community, we have released all the source code at https://FastBTS.github.io and an online prototype system at http://FastBTS.thucloud.com.

2 Understanding State-of-The-Art BTSes

2.1 Methodology

We measure a BTS using the following metrics: (1) *Test Accuracy* measures how well the result (r) reported by a BTS matches the ground-truth bandwidth R. We calculate the accuracy as $\frac{r}{R}$. In practice, we observe that all BTSes (including FastBTS) tend to underestimate the bottleneck bandwidth due to factors like TCP slow start and congestion control, so the accuracy values are less than 1.0. (2) *Test Duration* measures

[1] In this work, we focus on the downlink bandwidth test due to its importance to a typical Internet user compared to the uplink.

the time needed to perform a bandwidth test—from starting a bandwidth test to returning the test result. (3) *Data Usage* measures the consumed network traffic for a test. This metric is of particular importance to metered LTE and 5G links.

Obtaining ground truth. Measuring test accuracy requires ground-truth data. However, it is challenging to know all the ground-truth bandwidths for large measurements. We use best possible estimations for different types of access links:

- *Wired LANs for in-lab experiments.* We regard the (known) physical link bandwidth, with the impact of (our injected) cross traffic properly considered, as the ground truth.
- *Commercial residential broadband and cloud networks.* We collect the bandwidth claimed by the ISPs or cloud service providers from the service contract, denoted as T_C. We then verify T_C by conducting long-lived bulk data transfers (average value denoted as T_B) before and after a bandwidth test. In more than 90% of our experiments, T_B and T_C match, with their difference being less than 5%; thus, we regard T_C as the ground truth. Otherwise, we choose to use T_B.
- *Cellular networks (LTE and 5G).* Due to a lack of T_C and the high dynamics of cellular links, we leverage the results provided by `SpeedTest.net` as a baseline reference. Being the state-of-the-art BTS that owns a massive number of (∼12,000) test servers across the globe, `SpeedTest.net`'s results are widely considered as a close approximation to the ground-truth bandwidth [33, 36, 38, 41, 50, 73].

2.2 Analyzing Deployed BTSes

We study 20 deployed BTSes, including 18 widely-used, web-based BTSes and 2 Android 11 BTS APIs.[2] We run the 20 BTSes on three different PCs and four different smartphones listed in Table 1 (WiFiMaster and Android APIs are only run on smartphones). To understand the implementation of these BTSes, we jointly analyze: (1) the network traffic (recorded during each test), (2) the client-side code, and (3) vendors' documentation. A typical analysis workflow is as follows. We first examine the network traffic to reveal which server(s) the client interacts with during the test, as well as their interaction durations. We then inspect the captured HTTP(S) transactions to interpret the client's interactions with the server(s) such as server selection and file transfer. We also inspect client-side code (typically in JavaScript). However, this attempt may not always succeed due to code obfuscation used by some BTSes like SpeedTest. In this case, we use the Chrome developer tool to monitor the entire test process in the debug mode.

[2]The 18 web-based BTSes are ATTtest [2], BWP [5], CenturyLink [6], Cox [7], DSLReports [8], FAST [10], NYSbroadband [17], Optimum [19], SFtest [13], SpeakEasy [22], Spectrum [23], SpeedOf [24], SpeedTest [25], ThinkBroadband [28], Verizon [30], Xfinity [32], XYZtest [26], and WiFi-Master [31]. They are selected based on Alexa ranks and Google page ranks. In addition, we also study two BTS APIs in Android 11: `getLinkDownstreamBandwidthKbps` and `testMobileDownload`.

Device	Location	Network	Ground Truth
PC-1	U.S.	Residential broadband	100 Mbps
PC-2	Germany	Residential broadband	100 Mbps
PC-3	China	Residential broadband	100 Mbps
Samsung GS9	U.S.	LTE (60Mhz/1.9Ghz)	60–100 Mbps
Xiaomi XM8	China	LTE (40Mhz/1.8Ghz)	58–89 Mbps
Samsung GS10	U.S.	5G (400Mhz/28Ghz)	0.9–1.2 Gbps
Huawei HV30	China	5G (160Mhz/2.6Ghz)	0.4–0.7 Gbps

Table 1: Client devices used for testing the 20 BTSes. The test results are obtained from `SpeedTest.net`.

With the above efforts, we are able to "reverse engineer" the implementations of all the 20 BTSes.

Our analysis shows that a bandwidth test in these BTSes is typically done in three phases: (1) setup, (2) bandwidth probing, and (3) bandwidth estimation. In the setup phase, the BTS sends a list of candidate servers (based on the client's IP address or geo-location) to the client who then PINGs each candidate server over HTTP(S). Next, based on the servers' PING latency, the client selects one or more candidate servers to perform file transfer(s) to collect throughput samples. The BTS processes the samples and returns the result to the user.

2.3 Measurement Results

We select 9 (out of 20) representative BTSes for more in-depth characterizations, as listed in Table 2. These 9 selected BTSes well cover different designs (in terms of the key bandwidth test logic) of the remaining 11 ones. We deploy a large-scale testbed to comprehensively profile 8 representative BTSes, except Android API-A (we will discuss it separately). Our testbed is deployed on 108 geo-distributed VMs from multiple public cloud services providers (CSPs, including Azure, AWS, Ali Cloud, Digital Ocean, Vultr, and Tencent Cloud) as the client hosts. Note that we mainly employ VMs as client hosts because they are globally distributed and easy to deploy. Per their service agreements, the CSPs offer three types of access link bandwidths: 1 Mbps, 10 Mbps, and 100 Mbps (36 VMs each). The ground truth in Figure 2c is obtained according to the methodology in §2.1. We denote one *test group* as using one VM to run back-to-back bandwidth tests across all the 8 BTSes in a random order. We perform in one day 3,240 groups of tests, *i.e.,* 108 VMs × 3 different time-of-day (0:00, 8:00, and 16:00) × 10 repetitions.

We summarize our results in Table 2. We discover that all but one of the BTSes adopt *flooding-based* approaches to combat the test noises from a temporal perspective, leading to enormous data usage. Meanwhile, they differ in many aspects: (1) bandwidth probing mechanism, (2) bandwidth estimation algorithm, (3) connection management strategy, (4) server selection policy, and (5) server pool size.

2.4 Case Studies

We present our case studies of five major BTSes with the largest user bases selected from Table 2.

BTS	# Servers	Bandwidth Test Logic	Duration	Accuracy (Testbed / 5G)	Data Usage (Testbed / 5G)
TBB*	12	average throughput in all connections	8 s	0.59 / 0.31	42 MB / 481 MB
SpeedOf	116	average throughput in the last connection	8–230 s	0.76 / 0.22	61 MB / 256 MB
BWP	18	average throughput in the fastest connection	13 s	0.81 / 0.35	74 MB / 524 MB
SFtest	19	average throughput in all connections	20 s	0.89 / 0.81	194 MB / 2,013 MB
ATTtest	75	average throughput in all connections	15–30 s	0.86 / 0.53	122 MB / 663 MB
Xfinity	28	average all throughput samples	12 s	0.82 / 0.67	107 MB / 835 MB
FAST	∼1,000	average stable throughput samples	8–30 s	0.80 / 0.72	45 MB / 903 MB
SpeedTest	∼12,000	average refined throughput samples	15 s	0.96 / 0.92	150 MB / 1,972 MB
Android API-A	0	directly calculate using system configs	< 10 ms	NA / 0.09	0 / 0

Table 2: A brief summary of the 9 representative BTSes. "Testbed" and "5G" denote the large-scale cloud-based testbed and the 5G scenario, respectively. * means that WiFiMaster and Andriod API-B share the similar bandwidth test logic with ThinkBroadBand (TBB).

ThinkBroadBand [28]. The ThinkBroadBand BTS first selects a test server with the lowest latency to the client among its server pool. Then, it starts an 8-second bandwidth test by delivering a 20-MB file towards the client; if the file transfer takes less than 8 seconds, the test is repeated to collect more data points. After the 8 seconds, it calculates the average throughput (*i.e.,* data transfer rate) during the whole test process as the estimated bandwidth.

WiFiMaster [31]. WiFiMaster's BTS is largely the same as that of ThinkBroadBand. The main difference lies in the test server pool. Instead of deploying a dedicated test server pool, WiFiMaster exploits the CDN servers of a large Internet content provider (Tencent) for frequent bandwidth tests. It directly downloads fixed-size (∼47 MB) software packages as the test files and measures the average download speed as the estimated bandwidth.

Our measurements show that the accuracy of ThinkBroadBand and WiFiMaster is low. The accuracy is merely 0.59, because the single HTTP connection during the test can easily be affected by network spikes and link congestion which lead to significant underestimation. In addition, using the average throughput for bandwidth estimation cannot rule out the impact of slow start and thus requires a long test duration.

Android APIs [1, 3]. To cater to the needs of bandwidth estimation for bandwidth-hungry apps (*e.g.,* UHD videos and VR/AR) over 5G, Android 11 offers two "Bandwidth Estimator" APIs to "make it easier to check bandwidth for uploading and downloading content [1]".

API-A, `getLinkDownstreamBandwidthKbps`, statically calculates the access bandwidth by "taking into account link parameters (radio technology, allocated channels, *etc.*) [20]". It uses a pre-defined dictionary (`KEY_BANDWIDTH_STRING_ARRAY`) to map device hardware information to bandwidth values. For example, if the end-user's device is connected to the new-radio non-standalone mmWave 5G network, API-A searches the dictionary which records `NR_NSA_MMWAVE:145000,60000`, indicating that the downlink bandwidth is 145,000 Kbps and the uplink bandwidth is 60,000 Kbps. This API provides a static "start-up on idle" estimation [1]. We test the performance of API-A in a similar manner as introduced in §2.3 with the 5G phones in Table 1. The results show that API-A bears rather poor accuracy (0.09) in realistic scenarios.

API-B, `testMobileDownload`, works in a similar way as ThinkBroadBand. It requires the app developer to provide the test servers and the test files.

FAST [10] is an advanced BTS with a pool of about 1,000 test servers. It employs a two-step server selection process: the client first picks five nearby servers based on its IP address, and then PINGs these five candidates to select the latency-wise nearest server for the bandwidth probing phase.

FAST progressively increases the concurrency according to the client network condition during the test. The client starts with a 25-MB file over a single connection. When the throughput reaches 0.5 Mbps, a new connection is created to transfer another 25-MB file. Similarly, at 1 Mbps, a third connection is established. For each connection, when the file transfer completes, it repeatedly requests another 25-MB file (the concurrency level never decreases).

FAST estimates the bandwidth as follows. As shown in Figure 2a, it collects a throughput sample every 200 ms, and maintains a 2-second window consisting of 10 most recent samples. After 5 seconds, FAST checks whether the in-window throughput samples are stable: $S_{max} - S_{min} \leq 3\% \cdot S_{avg}$, where S_{max}, S_{min}, and S_{avg} correspond to the maximum, minimum, and average value across all samples in the window, respectively. If the above inequality holds, FAST terminates the test and returns S_{avg}. Otherwise, the test will continue until reaching a time limit of 30 seconds; at that time, the last 2-second window's S_{avg} will be returned to the user.

Unfortunately, our results show that the accuracy of FAST is still unsatisfactory. The average accuracy is 0.80, as shown in Table 2 and Figure 2c. We ascribe this to two reasons: (1) Though FAST owns ∼1,000 servers, they are mostly located in the US and Canada. Thus, FAST can hardly assign a nearby server to clients outside North America. In fact, FAST achieves relatively high average accuracy (0.92) when serving the clients in North America; however, it has quite low accuracy (0.74) when measuring the access link bandwidth of the clients in other places around the world. (2) We observe that FAST's window-based mechanism for early generation of the test result is vulnerable to throughput fluctuations. Under unstable network conditions, FAST can only use throughput samples in the last two seconds (rather than the entire

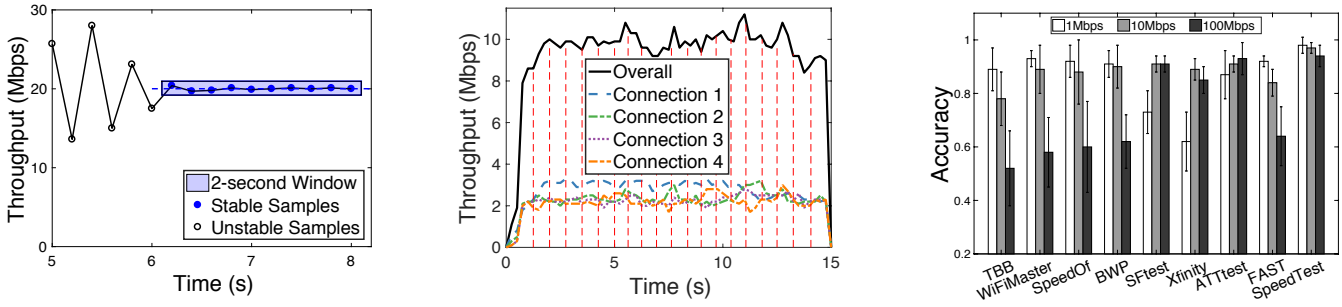


Figure 2: (a) Test logic of FAST. (b) Test logic of SpeedTest. (c) Test accuracy of nine commercial BTSes.

30-second samples) to calculate the test result.

SpeedTest [25] is considered the most advanced industrial BTS [33, 36, 41, 50, 73]. It deploys a pool of ∼12,000 servers. Similar to FAST, it also employs the two-step server selection process: it identifies 10 candidate servers based on the client's IP address, and then selects the latency-wise nearest from them. It also progressively increases the concurrency level: it begins with 4 parallel connections for quickly saturating the available bandwidth, and establishes a new connection at 25 Mbps and 35 Mbps, respectively. It uses a fixed file size of 25 MB and a fixed test duration of 15 seconds.

SpeedTest's bandwidth estimation algorithm is different from FAST's. During the bandwidth probing phase, it collects a throughput sample every 100 ms. Since the test duration is fixed to 15 seconds, all the 150 samples are used to construct 20 slices, each covering the same traffic volume, illustrated as the area under the throughput curve in Figure 2b. Then, 5 slices with the lowest average throughput and 2 slices with the highest average throughput are discarded. This leaves 13 slices remaining, whose average throughput is returned as the final test result. This method may help mitigate the impact of throughput fluctuations, but the two fixed thresholds for noise filtering could be deficient under diverse network conditions.

Overall, SpeedTest exhibits the highest accuracy (0.96) among the measured BTSes. A key contributing factor is its large server pool, as shown in §5.2.

3 Design of FastBTS

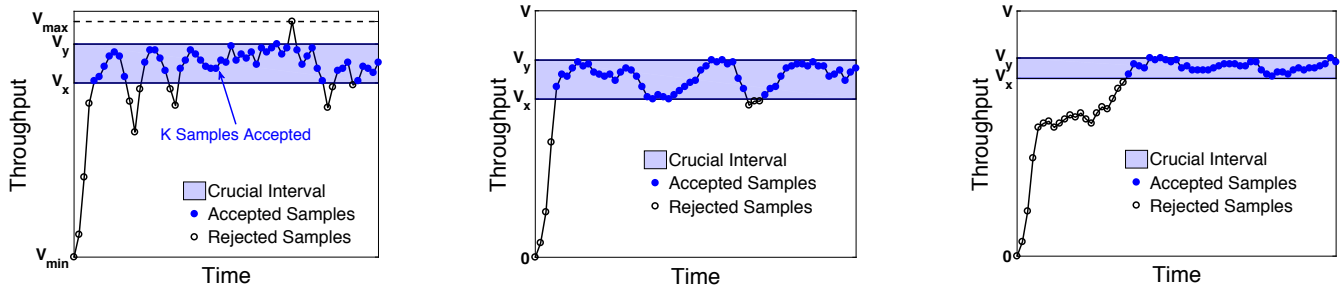
FastBTS is a fast and lightweight BTS with a fundamentally new design. FastBTS *accommodates and exploits* noises (instead of suppressing them) to significantly reduce the resource footprint and accelerate the tests, while retaining high test accuracy. The key technique of FastBTS is *fuzzy rejection sampling* which automatically identifies true samples that represent the target distribution and filters out false samples due to measurement noises, without apriori knowledge of the target distribution. Figure 1 shows the main components of FastBTS and the workflow of a bandwidth test.

- *Crucial Interval Sampling (CIS)* implements the acceptance rejection function of fuzzy rejection sampling. CIS is built upon a key observation based on our measurement study (see §2.3 and §2.4): *while the noise samples may be widely scattered, the desired bandwidth samples tend to concentrate within a narrow throughput interval*. CIS searches for a *dense* and *narrow* interval that covers the majority of the desirable samples, and uses computational geometry to drastically reduce the searching complexity.
- *Elastic Bandwidth Probing (EBP)* generates throughput samples that persistently[3] obey the distribution of the target bandwidth. We design EBP by optimizing BBR's bandwidth estimation algorithm [42] – different from BBR's static bandwidth probing policy, EBP reaches the target bandwidth much faster, while being non-disruptive.
- *Data-driven Server Selection (DSS)* selects the server(s) with the highest bandwidth estimation(s) through a data-driven model. We show that a simple model can significantly improve server selection results compared to the de-facto approach that ranks servers by round-trip time.
- *Adaptive Multi-Homing (AMH)* adaptively establishes multiple parallel connections with different test servers. AMH is important for saturating the access link when the last-mile access link is not the bottleneck, *e.g.,* 5G [67].

3.1 Crucial Interval Sampling (CIS)

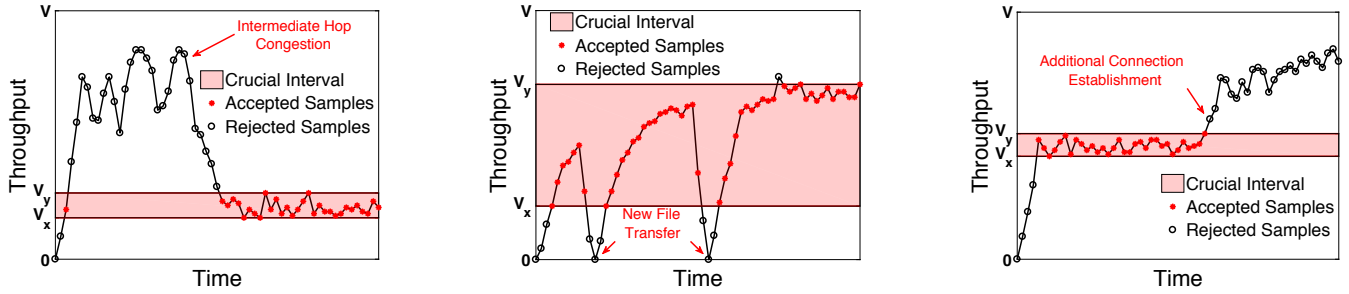
CIS is designed based on the key observation: while noise samples are scattered across a wide throughput interval, the desirable samples tend to concentrate within a narrow interval, referred to as the *crucial interval*. As shown in Figure 3, in each subfigure, although the crucial interval is narrow, it can cover the vast majority of the desirable samples. Thus, while the target distribution $T(x)$ is unknown, we can approximate $T(x)$ with the crucial interval. Also, as more noise samples accumulate, the test accuracy would typically increase as

[3] Here "persistently" means that a certain set (or range) of samples constantly recur to the measurement data during the test process [45].



(a) SpeedTest **(b)** ATTtest, BWP, XFinity, Android API-B **(c)** FAST

Figure 3: Common scenarios where the true samples fall in a crucial interval. In our measurement, > 96% cases fall into the three patterns.



(a) Slow start effect of sequential file transfer in SpeedOf and TBB **(b)** WiFiMaster when an intermediate hop suffers a temporary congestion **(c)** SpeedTest when another connection is established to saturate the access link

Figure 4: Pathological scenarios where the true samples are not persistently covered by the crucial interval (less than 4% in our measurements).

randomly scattered noise samples help better "contrast" the crucial interval, leading to its improved approximation.

Crucial Interval Algorithm. Based on the above insights, our designed bandwidth estimation approach for FastBTS aims at finding this crucial interval ($[V_x, V_y]$) that has both *a high sample density* and *a large sample size*. Assuming there are N throughput samples ranging from V_{min} to V_{max}, our aim is formulated as maximizing the *product of density and size*. We denote the size as $K(V_x, V_y)$, *i.e.,* the number of samples that fall into $[V_x, V_y]$. The density can be calculated as the ratio between $K(V_x, V_y)$ and $N' = N(V_y - V_x)/(V_{max} - V_{min})$, where N' is the "baseline" corresponding to the number of samples falling into $[V_x, V_y]$ *if* all N samples are uniformly distributed in $[V_{min}, V_{max}]$. To prevent a pathological case where the density is too high, we enforce a lower bound of the interval: $V_y - V_x$ should be at least L_{min}, which is empirically set to $(V_{max} - V_{min})/(N-1)$. Given the above, the objective function to be maximized is:

$$F(V_x, V_y) = Density \times Size = C \cdot \frac{K^2(V_x, V_y)}{V_y - V_x}, \qquad (1)$$

where $C = (V_{max} - V_{min})/N$ is a constant. Once the optimal $[V_x, V_y]$ is calculated, we can derive the bandwidth estimation by averaging all the samples falling into this interval.

FastBTS computes the crucial interval as bandwidth probing (§3.2) is in progress, which serves as the acceptance-rejection function (ARF) of rejection sampling. When a new sample is available, the server computes a crucial interval by maximizing Equation (1). It thus produces a series of intervals $[V_{x3}, V_{y3}]$, $[V_{x4}, V_{y4}]$, $\cdots$ where $[V_{xi}, V_{yi}]$ corresponds to the interval generated when the i-th sample is available.

Searching Crucial Interval with Convex Hull. We now consider how to actually solve the maximization problem in Equation (1). To enhance the readability, we use L to denote $V_y - V_x$, use K to denote $K(V_x, V_y)$, and let the maximum value of $F(V_x, V_y)$ be F_{max}, which lies in $(0, \frac{C \cdot N^2}{L_{min}}]$.

Clearly, a naïve exhaustive search takes $O(N^2)$ time. Our key result is that this can be done much more efficiently in $O(N \log N)$ by strategically searching on a convex hull dynamically constructed from the samples. Our high-level approach is to perform a binary search for F_{max}. The initial midpoint is set to $\lfloor \frac{C \cdot N^2}{2 \cdot L_{min}} \rfloor$. In each binary search iteration, we examine whether the inequality $\frac{C \cdot K^2}{L} - m \geq 0$ holds for any interval(s), where $0 < m \leq F_{max}$ is the current midpoint. Based on the result, we adjust the midpoint and continue with the next iteration.

We next see how each iteration is performed exactly. Without loss of generality, we assume that the throughput samples are sorted in ascending order. Suppose we choose the i-th and j-th samples ($i < j$) from the N sorted samples as the end-

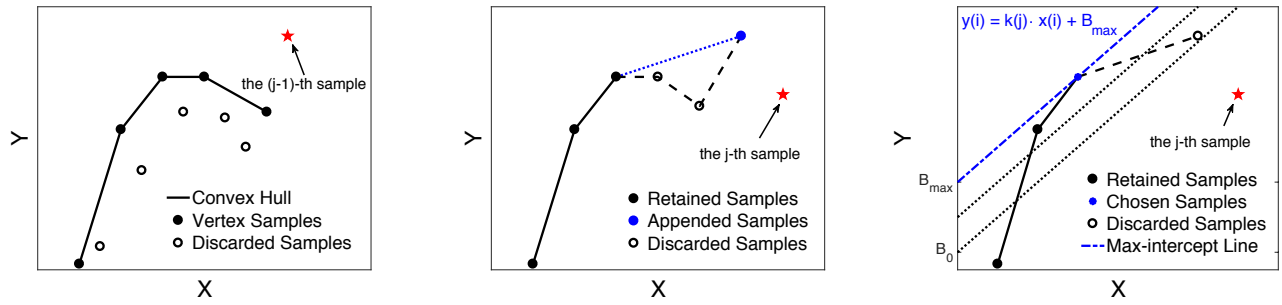


Figure 5: (a) The current convex hull under transformed coordinates, where the axes X and Y correspond to $x(i)$ and $y(i)$ in Equation (4) respectively. (b) Updating the convex hull with the $(j-1)$-th sample. (c) Searching for the max-intercept line.

points of the interval $[V_i, V_j]$. Then the inequality $\frac{C \cdot K^2}{L} - m \geq 0$ can be transformed as:

$$\frac{C \cdot (j-i+1)^2}{V_j - V_i} - m \geq 0. \tag{2}$$

We further rearrange it as:

$$i^2 - 2i + \frac{m}{C} V_i - 2ij \geq \frac{m}{C} V_j - 2j - j^2 - 1. \tag{3}$$

It is not difficult to discover that the right side of the inequality is only associated with the variable j, while the left side just relates to the variable i except the term $-2ij$. Therefore, we adopt the following coordinate conversion:

$$\begin{cases} k(j) = 2j \\ b(j) = \frac{m}{C} V_j - 2j - j^2 - 1 \\ x(i) = i \\ y(i) = i^2 - 2i + \frac{m}{C} V_i \end{cases} \tag{4}$$

With the above-mentioned coordinate conversion, the inequality (3) can be transformed as: $y(i) - k(j) \cdot x(i) \geq b(j)$. Then, determining whether the inequality holds for at least one pair of (i, j) is equivalent to finding the maximum of $f(i) = y(i) - k(j) \cdot x(i)$ for each $1 < j \leq N$.

As depicted in Figure 5a, we regard $\{x(i)\}$ and $\{y(i)\}$ as coordinates of N points on a two-dimensional plane (these points do not depend on j). It can be shown using the linear programming theory that for any given j, the largest value of $f(i)$ always occurs at a point that is on the convex hull formed by $(x(i), y(i))$. This dictates an algorithm where for each $1 < j \leq N$, we check the points on the convex hull to find the maximum of $f(i)$.

Since i must be less than j, each time we increment j (the outer loop), we progressively add one point $(x(j-1), y(j-1))$ to the (partial) convex hull, which is shown in Figure 5b. Then among all existing points on the convex hull, we search backward from the point with the largest $x(i)$ value to the smallest $x(i)$ to find the maximum of $f(i)$, and stops searching when $f(i)$ starts to decrease since the points are on the convex hull (the inner loop).

As demonstrated in Figure 5c, an analytic geometry explanation of this procedure is to determine a line with a fixed slope $y = k(j)x + B$, *s.t.* the line intersects with a point on the convex and the intercept B is maximized, and the maximized intercept corresponds to the maximum of $f(i)$.

Also, once the maximum of $f(i)$ is found at $(x(i'), y(i'))$ for a given j, all points that are to the right of $(x(i'), y(i'))$ can be removed from the convex hull – they must not correspond to the maximum of $f(i)$ for all $j' > j$. This is because (1) the slope of the convex hull's edge decreases as i increases, and (2) $k(j)$ increases as j increases. Therefore, using amortized analysis, we can show that in each binary search iteration, the overall processing time for all points is $O(N)$ as j grows from 1 to N. This leads to an overall complexity of $O(N \log N)$ for the whole algorithm.

Fast Result Generation. FastBTS selects a group of samples that well fit $T(x)$ as soon as possible while ensuring data reliability. Given two intervals $[V_{xi}, V_{yi}]$ and $[V_{xj}, V_{yj}]$, we regard their similarity as the Jaccard Coefficient [60]. FastBTS then keeps track of the similarity values of consecutive interval pairs *i.e.,* $S_{3,4}, S_{4,5}, \ldots$ If the test result stabilizes, the consecutive interval pairs' similarity value will keep growing from a certain value β, satisfying $\beta \leq S_{i,i+1} \leq \cdots \leq S_{i+k,i+k+1} \leq 1$. If the above sequence is observed, FastBTS determines that the result has stabilized and reports the bottleneck bandwidth as the average value of the throughput samples belonging to the most recent interval. The parameters β and k pose a tradeoff between accuracy and cost in terms of test duration and data traffic. Specifically, increasing β and k can yield a higher test accuracy while incurring a longer test duration and more data usage. Currently, we empirically set β=0.9 and k=2, which are found to well balance the tradeoff between the test duration and accuracy. Nevertheless, when dealing with those relatively rare cases that are not covered by this paper, BTS providers are recommended

to do pre-tests in order to find the suitable parameter settings before putting CIS mechanism into actual use.

Solutions to Caveats. There do exist some "irregular" bandwidth graphs in prior work [37, 48, 56] where CIS may lose efficacy. For instance, due to in-network mechanisms like data aggregation and multi-path scheduling, the millisecond-level throughput samples can vary dramatically (*i.e.,* sometimes the throughput is about to reach the link capacity, and sometimes the throughput approaches zero). To mitigate this issue, we learn from some BTSes (*e.g.,* SpeedTest) and use a relatively large time interval (50 ms) to smooth the gathered throughput samples. However, even with smoothed samples, it is still possible that CIS may be inaccurate if the actual access bandwidth is outside the crucial interval, or the interval becomes too wide to give a meaningful bandwidth estimation. In our experiences, such cases are rare (less than 4% in our measurements in §2.3). However, to ensure that our tests are stable in all scenarios, we design solutions to those pathological cases.

Figure 4 shows all types of the pathological cases of CIS we observe, where $P(x)$ deviates from $T(x)$ over time. FastBTS leverages three mechanisms to resolve these cases: (1) elastic bandwidth probing (§3.2) reaches bottleneck bandwidth in a short time, effectively alleviating the impact of slow-start effect in Figure 4a; (2) data-driven server selection (§3.3) picks the expected highest-throughput server(s) for bandwidth tests, minimizing the requirement of additional connection(s) in Figure 4c; (3) adaptive multi-homing (§3.4) establishes concurrent connections with different servers, avoiding the underestimations in Figures 4b and 4c. We will discuss these mechanisms in §3.2 – §3.4.

3.2 Elastic Bandwidth Probing (EBP)

In rejection sampling, $P(x)$ determines the boundary of $T(x)$. A bandwidth test measures $P(x)$ using bandwidth probing based on network conditions. It shares a similar principle as congestion control at the test server's transport layer—the goal is to accommodate diverse noises over the live Internet, while saturating the bandwidth of the access link. FastBTS employs BBR [42], an advanced congestion control algorithm, as a starting point for probing design. Specifically, FastBTS uses BBR's built-in bandwidth probing for bootstrapping.

On the other hand, bandwidth tests have different requirements compared with congestion control. For example, congestion control emphasizes stable data transfers over a long period, while a BTS focuses on obtaining accurate link capacity as early as possible with the lowest data usage. Therefore, we modify and optimize BBR to support bandwidth tests.

BBR Prime. BBR is featured by two key metrics: bottleneck bandwidth $BtlBw$ and round-trip propagation time RT_{prop}. It works in four phases: Startup, Drain, ProbeBW (probing the bandwidth), and ProbeRTT. A key parameter *pacing_gain* (*PG*) controls TCP pacing so that the capacity of a network path can be fully utilized while the queuing delay is minimized. BBR multiplies its measured throughput by *PG* to determine the data sending rate in the subsequent RTT. After a connection is established, BBR enters the Startup phase and exponentially increases the sending rate (*i.e.,* $PG = \frac{2}{ln2}$) until the measured throughput does not increase further, as shown in Figure 6. At this point, the measured throughput is denoted as B_0 and a queue is already formed at the bottleneck of the network path. Then, BBR tries to Drain it by reducing *PG* to $\frac{ln2}{2} < 1$ until there is expected to be no excess in-flight data. Afterwards, BBR enters a (by default) 10-second ProbeBW phase to gradually probe $BtlBw$ in a number of cycles, each consisting of 8 RTTs with $PGs = \{\frac{5}{4}, \frac{3}{4}, 1, 1, 1, 1, 1, 1\}$. We plot in Figure 6 four such cycles tagged as ①②③④. Finally (10 seconds later), the *maximum* value of the measured throughput samples is taken as the network path's $BtlBw$ and BBR enters a 200-ms ProbeRTT phase to estimate RT_{prop}.

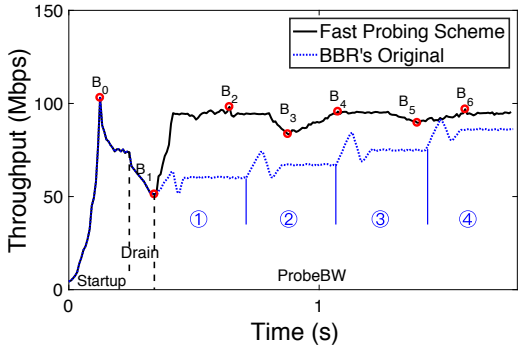


Figure 6: Elastic bandwidth probing vs. BBR's original scheme.

Limitations of BBR. Directly applying BBR's $BtlBw$-based probing method to BTSes is inefficient. First, as illustrated in Figure 6 (where the true $BtlBw$ is 100 Mbps), BBR's $BtlBw$ probing is conservative, making the probing process unnecessarily slow. A straightforward idea is to remove the 6 RTTs with $PG = 1$ in each cycle. Even with that, the probing process is still inefficient when the data (sending) rate is low. Second, when the current data rate (*e.g.,* 95 Mbps) is close to the true $BtlBw$ (*e.g.,* 100 Mbps), using the fixed PG of $\frac{5}{4}$ causes the data rate to far overshoot its limit (*e.g.,* to 118.75 Mbps). This may not be a severe issue for data transfers, but may significantly slow down the convergence of $BtlBw$ and thus lengthen the test duration. Third, BBR takes the maximum of all throughput samples in each cycle as the estimated $BtlBw$. The simple maximization operation is vulnerable to outliers and noises (this is addressed by CIS in §3.1).

Elastic Data-rate Pacing. We design *elastic pacing* to make bandwidth probing faster, more accurate, and more adaptive. Intuitively, when the data rate is low, it ramps up quickly to reduce the probing time; when the data rate approaches the estimated bottleneck bandwidth, it performs fine-grained probing by reducing the step size, towards a smooth convergence. This is in contrast to BBR's static probing policy.

We now detail our method. As depicted in Figure 6, once entering the ProbeBW phase, we have recorded B_0 and B where B_0 is the peak data rate measured during the Startup phase and B is the current data rate. Let the ground-truth bottleneck bandwidth be B_T. Typically, B_0 is slightly higher than B_T due to queuing at the bottleneck link in the end of the Startup phase (otherwise it will not exit Startup); also, B is lower than B_T due to the Drain phase. We adjust the value of B by controlling the *pacing gain* (PG) of the data sending rate, but the pivotal question here is the adjustment policy, *i.e.,* how to make B approach B_T quickly and precisely.

Our idea is inspired by the spring system [51] in physics where the restoring force of a helical spring is proportional to its elongation. We thus regard PG as the restoring force, and B's deviation from B_0 as the elongation (we do not know B_T so we approximate it using B_0). Therefore, the initial PG is expected to grow to:

$$PG_{grow-0} = (PG_m - PG_0) \times \left(1 - \frac{B}{B_0}\right) + PG_0, \quad (5)$$

where $1 - \frac{B}{B_0}$ denotes the normalized distance between B_0 and B, and PG_0 represents the default value (1.0) of PG. We set the upper bound of PG (PG_m) as $\frac{2}{ln2}$ that matches BBR's PG in the (most aggressive) Startup phase. As Equation (5) indicates, the spring system indeed realizes our idea: it increases the data rate rapidly when B is well below B_0, and cautiously reduces the probing step as B grows.

To accommodate a corner scenario where B_0 is lower than B_T ($< 1\%$ cases in our experiments), we slightly modify Equation (5) to allow B to overshoot B_0 marginally:

$$PG_{grow-0} = max\{(PG_m - PG'_0) \times \left(1 - \frac{B}{B_0}\right) + PG'_0, PG'_0\}, \quad (6)$$

where PG'_0 equals $1 + \varepsilon$, with ε being empirically set to 0.05.

When the data rate overshoots the bottleneck bandwidth (*i.e.,* the number of in-flight bytes exceeds the bandwidth-delay product), we reduce the data rate to suppress the excessive in-flight bytes. This is realized by inverting PG to $PG_{drop-0} = \frac{1}{PG_{grow-0}}$. This process continues until no excessive in-flight data is present. At that moment, B will again drop below B_T, so we start a new cycle to repeat the aforementioned data rate growth process.

To put things together, in our design, the ProbeBW phase consists of a series of cycles each consisting of only two stages: *growth* and *drop*. Each stage has a variable number of RTTs, and the six RTTs with $PG = 1$ in the original BBR algorithm are removed. The transitions between the two stages are triggered by the formation and disappearance of excessive in-flight bytes (*i.e.,* the queueing delay). In the i-th cycle, the PGs for the two stages are:

$$\begin{cases} PG_{grow-i} = max\{(PG_m - PG'_0) \times \left(1 - \frac{B}{B_i}\right) + PG'_0, PG'_0\}, \\ PG_{drop-i} = \frac{1}{PG_{grow-i}}, \end{cases} \quad (7)$$

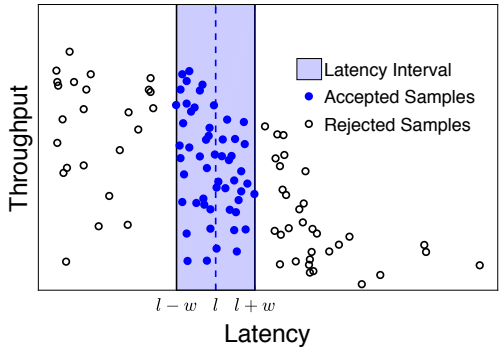


Figure 7: Data-driven server selection that takes both historical latency and throughput information into account.

where B is the data rate at the beginning of a growth stage, and B_i is the peak rate in the previous cycle's growth stage. Interestingly, by setting $B_0 = +\infty$, we can make the growth and drop stages identical to BBR's Startup and Drain phases, respectively. Our final design thus only consists of a single phase (ProbeBW) with two stages. The ProbeRTT phase is removed because it does not help our bandwidth probing.

Compared with traditional bandwidth probing mechanisms, EBP can saturate available bandwidth more quickly as it ramps up the sending rate when the current rate is much lower than the estimated bandwidth. Meanwhile, when the sending rate is about to reach the estimated bandwidth, EBP carefully increases the rate in order to be less aggressive to other flows along the path than other bandwidth probing mechanisms.

3.3 Data-driven Server Selection (DSS)

FastBTS includes a new server selection method. We find that selecting the test server(s) with the lowest PING latency, widely used in existing BTSes, is ineffective. Our measurement shows that latency and the available bandwidth are not highly correlated—the servers yielding the highest throughput may not always be those with the lowest PING latency.

FastBTS takes a *data-driven approach* for server selection (DSS): each test server maintains a database (model) containing {latency, throughput} pairs obtained from the setup and bandwidth probing phases of past tests. Then in a new setup phase, the client still PINGs the test servers, while each server returns an expected throughput value based on the PING latency by looking up the database. The client will then rank the selected server(s) based on their expected throughput values.

As demonstrated in Figure 7, the actual DSS algorithm is conceptually similar to CIS introduced in §3.1, whereas we empirically observe that only considering the density can yield decent results. Specifically, given a latency measurement l, the server searches for a width w that maximizes the density defined as $K(l,w)/2w$, where $K(l,w)$ denotes the number of latency samples falling in the latency interval $[l-w, l+w]$. The expected throughput is calculated as an average of all

samples in $[l-w, l+w]$. In addition, the server also returns the maximum throughput (using the 99-percentile value) belonging to $[l-w, l+w]$ to the client. Both values will be used in the bandwidth probing phase (§3.4). During bootstrapping when servers have not yet accumulated enough samples, the client can fallback to the traditional latency-based selection strategy. To keep their databases up-to-date, servers can maintain only most recent samples.

3.4 Adaptive Multi-Homing (AMH)

For high-speed access networks like 5G, the last-mile access link may not always be the bottleneck. To saturate the access link, we design an adaptive multi-homing (AMH) mechanism to dynamically adjust the concurrency level, *i.e.,* the number of concurrent connections between the servers and client.

AMH starts with a single connection to cope with possibly low-speed access links. For this single connection C_1, when CIS (§3.1) has accomplished using the server S_1 (the highest-ranking server, see §3.3), the reported bottleneck bandwidth is denoted as BW_1. At this time, the client establishes another connection C_2 with the second highest-ranking server S_2 while retaining C_1. C_2 also works as described in §3.2. Note we require S_1 and S_2 to be in different ASes to minimize the likelihood that S_1 and S_2 share the same Internet-side bottleneck. Moreover, we pick the server with the second highest bandwidth estimation as S_2 to saturate the client's access link bandwidth with the fewest test servers. After that, we view C_1 and C_2 together as an "aggregated" connection, with its throughput being $BW_2 = BW_{2,1} + BW_{2,2}$, where $BW_{2,1}$ and $BW_{2,2}$ are the real-time throughput of C_1 and C_2 respectively.

By monitoring $BW_{2,1}$, $BW_{2,2}$, and BW_2, FastBTS applies intelligent throughput sampling and fast result generation (§3.1) to judge whether BW_2 has become stable. Once BW_2 stabilizes, AMH determines whether the whole bandwidth test process should be terminated based on the relationship between BW_1 and $BW_{2,1}$. If for C_1 the bottleneck link is not the access link, $BW_{2,1}$ should have a value similar to or higher than BW_1 (assuming the unlikeliness of C_1 and C_2 sharing the same Internet-side bottleneck [53]). In this case, the client establishes another connection with the third highest-ranking server S_3 (with a different AS), and repeats the above process (comparing $BW_{3,1}$ and BW_1 to decide whether to launch the fourth connection, and so on). Otherwise, if $BW_{2,1}$ exhibits a noticeable decline (empirically set to $> 5\%$) compared to BW_1, we regard that C_1 and C_2 saturate the access link and incur cross-flow contention. In this case, the client stops probing and reports the access bandwidth as $max(BW_1, BW_2)$.

4 Implementation

As shown in Figure 1, we implement *elastic bandwidth probing* (EBP) and *crucial interval sampling* (CIS) on the server side, because EBP works at the transport layer and thus requires OS kernel modifications, and CIS needs to get fine-grained throughput samples from EBP in real time. We implement EBP and CIS in C and Node.js, respectively.

We implement *data-driven server selection* (DSS) and *adaptive multi-homing* (AMH) on the client side. End users can access the FastBTS service through REST APIs. We implement DSS and AMH in JavaScript to make them easy to integrate with web pages or mobile apps.

The test server is built on CentOS 7.6 with the Linux kernel version of 5.0.1. As mentioned in §3.2, we develop EBP by using BBR as the starting point. Specifically, we implement the calculation of *pacing_gain* according to Equation (7) by modifying the `bbr_update_bw` function; we also modify `bbr_set_state` and `bbr_check_drain` to alter BBR's original cycles in the ProbeBW phase, so as to realize EBP's two-stage cycles. EBP is implemented as a loadable kernel module. CIS is a user-space program. To efficiently send in-situ performance statistics including throughput samples, $Btlbw$, and RT_{prop} from EBP to CIS (both EBP and CIS are implemented on the server side in C and Node.js), we use the Linux Netlink Interface in `netlink.h` to add a raw socket and a new packet structure `bbr_info` that carries the above performance information. The performance statistics are also sent to the client by piggybacking with probing traffic, allowing users to examine the real-time bandwidth test progress.

5 Evaluation

5.1 Experiment Setup

We compare FastBTS with the 9 state-of-the-art BTSes studied in §2. For fair comparisons, we re-implement all the BTSes based on our reverse engineering efforts and use the same setup for all these re-implemented BTSes. To do so, we built the following testbeds.

Large-scale Testbed. We deploy a total of 30 test servers on 30 VMs across the globe (North America, South America, Asia, Europe, Australia, and Africa) with the same configurations (dual-core Intel CPU@2.5 GHz, 8-GB DDR memory, and 1.5+ Gbps outgoing bandwidth). The size of the server pool (30) is on par with 5 out of the 9 BTSes but is smaller than those of FAST and SpeedTest (Table 2), which we assume is a representative server pool size adopted by today's commercial BTSes. We deploy 100+ clients including 3 PCs, 4 smartphones, and 108 VMs (the same as those adopted in §2). For a fair comparison with FastBTS, we replicate the 9 other popular BTSes: SpeedOf, BWP, SFtest, ATTtest, Xfinity, FAST, SpeedTest, TBB, and Android API-A (see §2.3) and deploy them on the 30 test servers and the 100+ clients. We deploy API-A (Android specific) on 4 phones.

Tested Networks. We conduct extensive evaluations under heterogeneous networks. We detail their setups below.

- *Residential Broadband.* We deploy three PCs located in China, U.S., and Germany (Table 1). All the PCs' access links are 100 Mbps residential broadband. The three clients

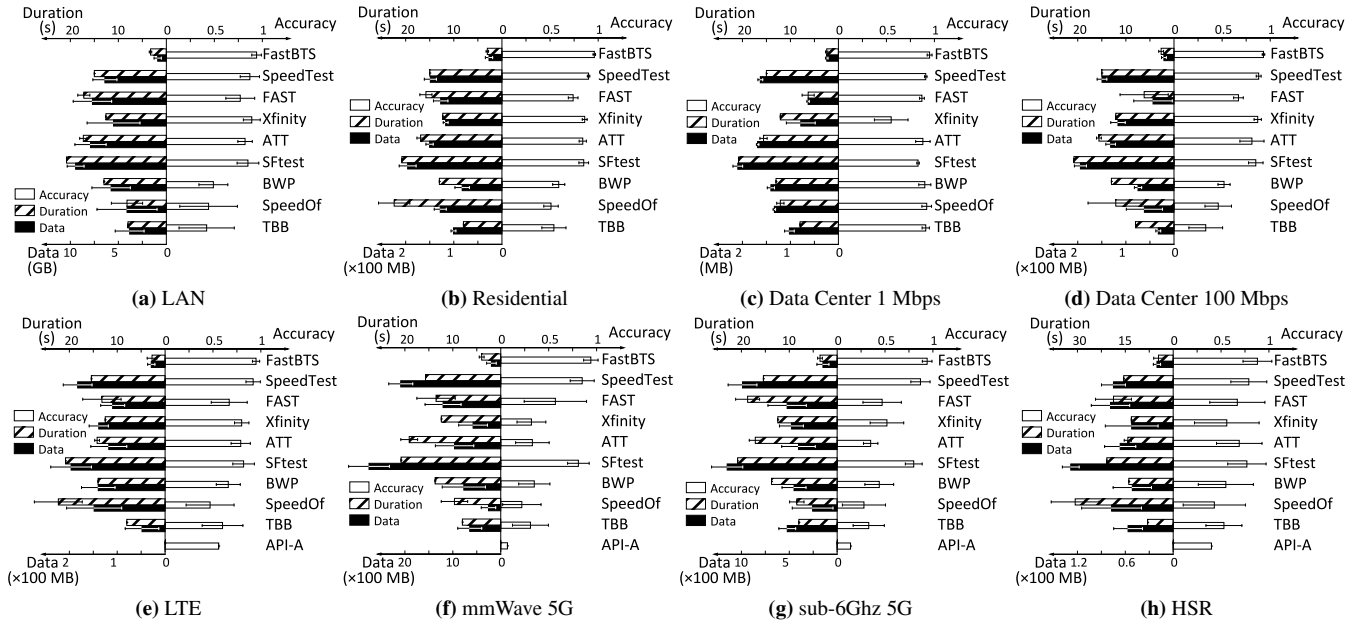


Figure 8: Duration and test accuracy of FastBTS, compared with 9 BTSes under various networks. "API-A" refers to Android API-A (§2.4).

communicate with (a subset of) the aforementioned 30 test servers to perform bandwidth tests. We perform in one day 90 groups of tests, consisting of 3 clients × 3 different time-of-day (0:00, 8:00, and 16:00) × 10 repetitions.

- *Data Center Networks.* We deploy 108 VMs belonging to different commercial cloud providers as the clients (§2.3). We perform a total number of 108 VMs × 3 time-of-day × 10 repetitions = 3,240 groups of tests.
- *mmWave 5G* experiments were conducted at a downtown street in a large U.S. city, with a distance from the phone (Samsung GS10, see Table 1) to the base station of 30m. This is a typical 5G usage scenario due to the small coverage of 5G base stations. The phone typically has line-of-sight to the base station unless being blocked by passing vehicles. The typical downlink throughput is between 0.9 and 1.2 Gbps. We perform in one day 120 groups of tests, consisting of 4 clients × 3 time-of-day × 10 repetitions.
- *Sub-6Ghz 5G* experiments were conducted in a Chinese city using an HV30 phone over China Mobile. The setup is similar to that of mmWave. We run 120 groups of tests.
- *LTE* experiments were conducted in both China (a university campus) and U.S. (a large city's downtown area) using XM8 and GS9, respectively, each with 120 groups of tests.
- *HSR Cellular Access.* We also perform tests on high-speed rail (HSR) trains. We take the Beijing-Shanghai HSR line (peak speed of 350 km/h) with two HV30 phones. We measure the LTE bandwidth from the train. We run 2 clients × 50 repetitions = 100 groups of tests.
- *LAN.* Besides the deployment of 30 VMs, we also create an in-lab LAN testbed to perform controlled experiments, where we can craft background traffic. The testbed consists of two test servers (S_1, S_2) and two clients (C_1, C_2), each equipping a 10 Gbps NIC. They are connected by a commodity switch with a 5 Gbps forwarding capability, thus being the bottleneck. When running bandwidth tests on this testbed, we maintain two parallel flows: one 1 Gbps background flow between S_1 and C_1, and a bandwidth test flow between S_2 and C_2.

We use the three metrics described in §2.1 to assess BTSes: test duration, data usage, and accuracy. Also, the methodology for obtaining the ground truth is described in §2.1.

5.2 End-to-End Performance

LAN and Residential Networks. As shown in Figure 8a and 8b, FastBTS yields the highest accuracy (0.94 for LAN and 0.96 for residential network) among the 9 BTSes, whose accuracy lies within 0.44–0.89 for LAN, and 0.51–0.9 for residential network. The average test duration of FastBTS for LAN and residential network is 3.4 and 3.0 seconds respectively, which are 2.4–7.4× shorter than the other BTSes. The average data usage of FastBTS is 0.9 GB for LAN and 27 MB for residential network, which are 3.1–10.5× less than the other BTSes. The short test duration and small data usage are attributed to EBP (§3.2), which allows a rapid data rate increase when the current data rate is far lower than the bottleneck bandwidth, as well as fast result generation, which strategically trades off accuracy for a shorter test duration.

Data Center Networks. Figure 8c and 8d show the performance of different BTSes in CSPs' data center networks with

the bandwidth of $\{1, 100\}$ Mbps (per the CSPs' service agreements). FastBTS outperforms the other BTSes by yielding the highest accuracy (0.94 on average), the shortest test duration (2.67 seconds on average), and the smallest data usage (21 MB on average for 100-Mbps network). In contrast, the other BTSes' accuracy ranges between 0.46 and 0.91; their test duration is also much longer, from 6.2 to 20.8 seconds, and they consume much more data (from 44 to 194 MB) compared to FastBTS. In particular, we find that on low-speed network (1 Mbps), some BTSes such as Xfinity and SFtest establish too many parallel connections. This leads to poor performance due to the excessive contention across the connections. FastBTS addresses this issue through AMH (§3.4) that adaptively adjusts the concurrency level according to the network condition. The results of networks with 10 Mbps bandwidth are similar to those of 100 Mbps networks.

LTE and 5G Networks. We evaluate the BTSes' performance on commercial LTE and 5G networks (both mmWave and sub-6Ghz for 5G). Over LTE, as plotted in Figure 8e, FastBTS owns the highest accuracy (0.95 on average), the smallest data usage (28.23 MB on average), and the shortest test duration (2.73 seconds on average). The other 9 BTSes are far less efficient: 0.62–0.92 for average accuracy, 41.8 to 179.3 MB for data usage, and 7.1 to 20.8 seconds for test duration. For instance, we discover that FAST bears a quite low accuracy (0.67) because its window-based mechanism is very vulnerable to throughput fluctuations in LTE. SpeedTest, despite having a decent accuracy (0.92), incurs quite high data usage (166.3 MB) since it fixes the bandwidth test duration to 15 seconds regardless of the stability of the network.

Figure 8f shows the results for mmWave 5G. It is also encouraging to see that FastBTS outperforms the 9 other BTSes across all three metrics (0.94 vs. 0.07–0.87 for average accuracy, 194.7 vs. 101–2,749 MB for data usage, and 4.0 vs. 8.9–26.2 seconds for test duration). Most of the BTSes have low accuracy (< 0.6); Speedtest and SFtest bear relatively high accuracy (0.81 and 0.85). However, the high data usage issue due to their flooding nature is drastically amplified in mmWave 5G. For example, Speedtest incurs very high data usage—up to 2,087 MB per test. The data usage for FAST is even as high as 2.75 GB. FastBTS addresses this issue through the synergy of its key features for fuzzy rejection sampling such as EBP and CIS. We observe similar results in the sub-6Ghz 5G experiments as shown in Figure 8g.

HSR Cellular Access. We also benchmark the BTSes on an HSR train running at a peak speed of 350km/h from Beijing to Shanghai. As shown in Figure 8h, the accuracy of all 10 BTSes decreases. This is attributed to two reasons. First, on HSR trains, the LTE bandwidth is highly fluctuating because of, *e.g.,* frequent handovers caused by high mobility and the contention traffic from other passengers. Second, given such fluctuations, performing bulk transfer before and after a bandwidth test can hardly capture the ground truth bandwidth, which varies significantly during the test. Nevertheless, compared to the other 9 BTSes, FastBTS still achieves the best performance (0.88 vs. 0.26–0.84 for average accuracy, 20.3 vs. 16–155 MB for data usage, and 4.6 vs. 10.6–32.4 seconds for test duration). The test duration is longer than the stationary scenarios because under high mobility, network condition fluctuation makes crucial intervals converge slower.

5.3 Individual Components

We evaluate the benefits of each component of FastBTS by incrementally enabling one at a time. When EBP is not enabled, we use BBR. When CIS is not enabled, we average the throughput samples to calculate the test result. When DSS is not enabled, test server(s) are selected based on PING latency. When AMH is not enabled, we apply SpeedTest's (single-homing) connection management logic.

Bandwidth Probing Schemes. We compare BBR-based FastBTS and SpeedTest under mmWave 5G. The average data usage of BBR per test (735 MB) is 65% less than that of our replicated SpeedTest (2,087 MB). Meanwhile, the accuracy of BBR slightly reduces from 0.85 to 0.81. The results indicate that BBR's $BtlBw$ estimation mechanism better balances the tradeoff between accuracy and data usage compared to flooding-based methods. We next compare BBR and EBP. We find that our EBP brings further improvements over BBR: under mmWave, EBP achieves an average data usage of 419 MB (compared 735 MB in BBR, a 42% reduction) and an average accuracy of 0.87 (compared to 0.81 in BBR, a 7% improvement). The advantages of EBP come from its elastic PG setting mechanism that is critical for adaptive bandwidth probing. Next, we compare BBR and EBP under data center networks. As shown in Figure 9a, compared to BBR, EBP reduces the average test duration by 40% (from 11.3 to 6.6 seconds) and the data usage by 38% (from 107 to 66 MB), while improving the average accuracy by 6%.

Probing Intrusiveness. We evaluate the probing intrusiveness of vanilla BBR and EBP using the LAN testbed (§5.1). Recall that we simultaneously run a 1 Gbps background flow and the bandwidth test flow that shares a 5 Gbps bottleneck at the switch. Ideally, a BTS should measure the bottleneck bandwidth to be 4 Gbps without interfering with the background flow, whose average/stdev throughput is thus used as a metric to assess the intrusiveness of BBR and EBP. Our test procedure is as follows. We first run the background flow alone for 1 minute and measure its throughput as R_{Origin}= 1 Gbps. We then run BBR and EBP with the background flow and measure the average (standard deviation) of the background flow throughput as R_{BBR} (S_{BBR}) and R_{EBP} (S_{EBP}), respectively, during the test. We demonstrate the three groups' throughput samples with their timestamps normalized by BBR's test duration in Figure 9b. EBP incurs a much smaller impact on the background flow compared to BBR, with R_{EBP} and R_{BBR} measured to be 0.97 Gbps and 0.90 Gbps, respectively.

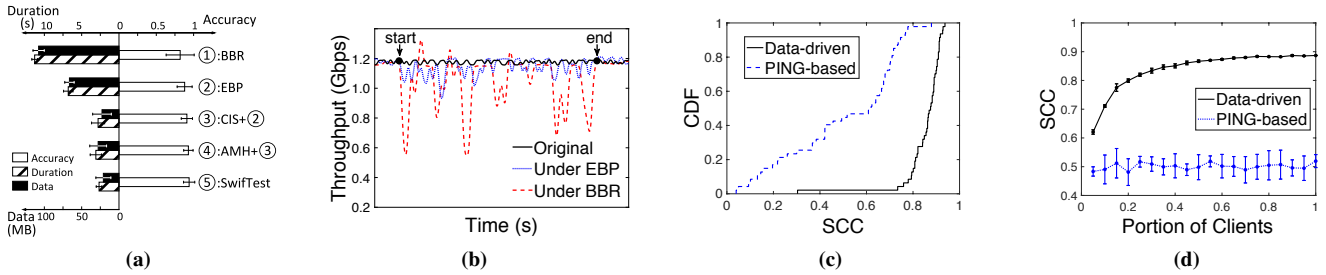


Figure 9: (a) Impact of individual modules of FastBTS in 100-Mbps data center networks. (b) Comparing intrusiveness between EBP and BBR. (c) Distributions of SCC_{gp} (PING-based) and SCC_{gd} (Data-driven) when P=20%; (d) P (portion of clients) vs. SCC_{gp} and SCC_{gd}.

Also, under EBP, the background flow's throughput variation is lower than under BBR: S_{EBP}/R_{EBP} and S_{BBR}/R_{BBR} are calculated to be 0.03 and 0.15, respectively. This suggests that when probing the bandwidth, EBP is less intrusive than BBR. We repeat the above test procedure under other settings, with the background flow's bandwidth varying from 0.5 to 4 Gbps, and observe consistent results. The lower intrusiveness of EBP compared with vanilla BBR probably lies in that when the sending rate is about to hit the access link bandwidth, EBP tends to carefully reclaim the available bandwidth; however, vanilla BBR still increases the sending rate with a fixed step, which is more aggressive than EBP in this scenario.

Crucial Interval Sampling (CIS). We further enable CIS. As shown in Figure 9a, by strategically removing outliers, CIS increases the average accuracy from 0.87 (EBP only) to 0.91. Due to the fast result generation mechanism, the test duration is reduced from 6.8 seconds to 2.8 seconds and the data usage is reduced by 2.8× (compared with EBP only).

We next compare CIS with the sampling approaches used by the other 9 BTSes (§5.1), which use a total of five bandwidth sampling algorithms because SFtest, ATTtest, and Xfinity employ the same trivial approach of simply averaging the throughput samples. To fairly compare them with CIS, we take a replay-based approach. Specifically, we select one "template" BTS from which we collect the network traces during the bandwidth probing phase; the time series of the aggregated throughput across all connections is then obtained from the traces and fed to all the sampling algorithms. We exclude SpeedOf and BWP from this experiment because they calculate the bandwidth based on the last or fastest connection that cannot be precisely reconstructed by our replay approach. We next show the results by using SpeedTest as the template BTS. The simplest algorithm (averaging) bears the lowest accuracy (0.81) because it is poor at eliminating the noises caused by, for example, TCP congestion control; the accuracy values of FAST, and SpeedTest are 0.82 and 0.84, respectively. In contrast, CIS owns the highest accuracy (0.91). This confirms the effectiveness of CIS's sampling approach. The efficiency and effectiveness of CIS lies in that, instead of incurring much redundancy in test duration and data usage to achieve a decent test accuracy, CIS keeps calculating the crucial interval of the gathered throughput samples. Once the crucial interval stabilizes, CIS immediately stops the test, thus significantly saving test duration and data usage.

Adaptive Multi-Homing (AMH). When AMH is further enabled, the test accuracy increases from 0.91 (EBP+CIS) to 0.93 (EBP+CIS+AMH), as shown in Figure 9a. Meanwhile, since testing over more connections takes additional time, AMH slightly lengthens the average test duration from 2.8 to 3.1 seconds, with the average data usage increased from 23 MB to 28 MB. We repeat the above experiments over mmWave 5G networks where the bottleneck is more likely to shift to the Internet side. The results show that AMH improves the average accuracy from 0.84 to 0.91, while incurring moderate overhead by increasing the average data usage from 148 MB to 206 MB and the average test duration from 3.3 to 4.1 seconds. The results suggest that AMH is essential for high-speed networks such as mmWave 5G.

Data-driven Server Selection (DSS). We employ *cross-validation* for a fair comparison between the PING-based method and DSS in three steps: (1) We do file transfers between every server and a randomly selected portion (P) of all clients to gather throughput samples. (2) Each client C runs a bandwidth test towards every server. In each test, the clients' historical test records (excluding the record of C) gathered in the previous step is utilized by each server to calculate the expected bandwidth, which is then returned to C. (3) Each client calculates three rankings of the servers based on the server-returned expected bandwidth: $Rank_g$, $Rank_p$, and $Rank_d$. $Rank_g$ refers to the server ranking based on the ground truth. $Rank_p$ is the ranking calculated based on PING latency; and $Rank_d$ is the ranking computed by DSS. We use the *Spearman Correlation Coefficient* (SCC [59]) to calculate the similarity SCC_{gp} between $Rank_g$ and $Rank_p$, as well as the similarity SCC_{gd} between $Rank_g$ and $Rank_d$.

The distributions of SCC_{gp} and SCC_{gd} when $P = 20\%$ are shown in Figure 9c. We find that SCC_{gd} is much higher than SCC_{gp} in terms of the median (0.81 *vs.* 0.63), average (0.80 *vs.* 0.50), and maximum (0.93 *vs.* 0.88) values. Further, Figure 9d shows that SCC_{gd} drops as P decreases; however, even

when P decreases to 5%, SCC_{gd} (0.62) is still 24% larger than SCC_{gp} (0.5). These results show that even with limited historical data, DSS works reasonably well. We enable DSS in our experiments of Figure 9a with $P = 20\%$. Compared to EBP+CIS+AMH, enabling DSS improves the average accuracy from 0.93 to 0.94; the test duration slightly reduces.

Overall Runtime Overhead. The client-side overhead of FastBTS is negligible based on our measurement on Samsung Galaxy S9, S10, Xiaomi M8, and Huawei Honor V30. On the server side, the incurred overhead is also low. When $Btlbw$ is 100 Mbps, the CPU overhead is measured to be lower than 5% (single core, tested on Intel CPU@2.5 GHz, 8-GB memory). The CPU overhead is only 12% when $Btlbw$ is 5 Gbps.

6 Related Work

Bandwidth Measurement. Bandwidth measurement is an essential component for many networked systems that empower many important applications and use cases [46, 61, 62, 75]. Apart from the BTSes described in §2, other bandwidth measurement methods mostly target specific types of networks (*e.g.,* datacenter [44], LTE [56, 71], and wireless [74, 77]) and require special support from the deployed infrastructure. For example, AuTO [44] conducts bandwidth estimation over DCTCP in data centers; it needs switch support to tag ECN marks on the data packets, and thus is challenging to be applied in WAN. Huang et al. [56] propose to deploy monitors inside the cellular core network for bandwidth measurement. Dischinger et al. [47] devise a bandwidth measurement tool which concurrently leverages multiple packet trains with different sending rates to measure the link bandwidth of residential broadband network.

While almost all commercial BTSes employ flooding-based methods to combat measurement noises, there exists quite a few non-flooding methods [55, 65, 68, 70] in academia, which indirectly infer the available bandwidth based on timing information of crafted packets (including packet pairs and packet trains). Unfortunately, these methods are highly sensitive to timing information, and thus can be easily disrupted by many factors like packet loss [54, 64], queueing [54], and data/ACK aggregation [64], especially in high-speed networks.

Designed as a generic network service for Internet users, FastBTS differs from and complements the above work. FastBTS targets at conducting fast and light bandwidth tests especially for high-speed wide-area networks (*e.g.,* 5G), significantly reducing data usage and test duration for clients. It does not require any hardware support at the client side. On the server side, we show that FastBTS requires a much smaller deployment to achieve the same level of effectiveness of existing large-scale commercial BTSes (*e.g.,* SpeedTest).

Congestion Control. FastBTS's elastic bandwidth probing is inspired by congestion control algorithms [63, 66, 79]. We categorize congestion control algorithms based on the congestion indicators: (1) *Loss-based CCs* (*e.g.,* BIC-TCP [76] and CUBIC [52]) which take packet loss as the indicator. They are vulnerable to bufferbloat and random losses [34]. (2) *Delay-based CCs* (*e.g.,* TCP FAST [58] and TCP Vegas [40]) which take transmission delay as the indicator. They are known to under-utilize the available bandwidth as the Internet latency is inherently noisy and fluctuating. (3) *Rate-based CCs* (*e.g.,* BBR [42], PCC [48] and PCC Vivace [49]) which directly estimate the available bandwidth and accordingly adjust data sending rate, typically via a feedback loop. We choose to design elastic bandwidth probing based on BBR, because BBR is mature with large-scale deployment on WAN [18], edge [27, 57], and cellular networks [35].

7 Concluding Remarks

We present FastBTS, a novel bandwidth testing system, to make bandwidth testing fast and light as well as accurate. By accommodating and exploiting the test noises, FastBTS achieves the highest level of accuracy among commercial BTSes, while significantly reducing data usage and test duration. Further, FastBTS only employs 30 servers, 2–3 orders of magnitude fewer than the state of the arts.

Despite the above merits, FastBTS still bears several limitations at the moment. First, when testing a client's uplink bandwidth, FastBTS requires extra deployment efforts (in particular a kernel module of EBP, as demonstrated in Figure 1) at the client side. Second, the performance of the data-driven server selection (DSS) mechanism can be affected by its cold start phase as well as the specific deployment of test servers. Third, when the selected test servers cannot saturate the client's downlink bandwidth, the adaptive multi-homing (AMH) mechanism may need several rounds to make the bandwidth probing process converge, thus leading to a relatively long test duration. We have been exploring practical ways to overcome these limitations.

Acknowledgements

We sincerely thank the anonymous reviewers for their valuable comments, and our shepherd Prof. Andreas Haeberlen for guiding us through the revision process. Also, we appreciate the generous help from Hongzhe Yang and Jiaxing Qiu in system deployment and text proofreading. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61902211, 61632020 and 61632013, and the Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] Add 5G capabilities to your app. https://developer.android.com/about/versions/11/features/5g.

[2] AT&T BTS. http://speedtest.att.com/speedtest/.

[3] BandwidthTest API in Android. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/tests/bandwidthtests/src/com/android/bandwidthtest/BandwidthTest.java.

[4] BTS Insights. https://www.speedtest.net/insights.

[5] BWP BTS. https://www.bandwidthplace.com/.

[6] Centurylink BTS. https://www.centurylink.com/home/help/internet/internet-speed-test.html/.

[7] Cox BTS. https://www.cox.com/residential/support/internet/speedtest.html.

[8] DSLReports. http://www.dslreports.com/speedtest/.

[9] Eighth Measuring Broadband America Fixed Broadband Report: A Report on Consumer Fixed Broadband Performance in the United States by (2018). Technical report, Federal Communications Commission.

[10] FAST BTS. https://fast.com/.

[11] Home Network Tips for the Coronavirus Pandemic. https://www.fcc.gov/home-network-tips-coronavirus-pandemic.

[12] How Coronavirus Affects Internet Usage and What You Can Do to Make Your Wi-Fi Faster. https://www.nbcnewyork.com/news/local/how-coronavirus-affects-internet-usage-and-what-you-can-do-to-make-your-wi-fi-faster/2332117/.

[13] HTML5 Speed Test by SourceForge. https://sourceforge.net/speedtest/.

[14] Measuring Broadband America Fixed Broadband Report (2016). Technical report, Federal Communications Commission.

[15] Measuring Broadband America Fixed Broadband Report: A Report on Consumer Fixed Broadband Performance in the US by (2014). Technical report, Federal Communications Commission.

[16] Nperf BTS. https://www.nperf.com/en/map/US/-/2420.ATT-Mobility/signal/?ll=37.59682400108367&lg=-109.44030761718751&zoom=8.

[17] NYSbroadband BTS. http://nysbroadband.speedtestcustom.com/.

[18] Optimizing HTTP/2 Prioritization with BBR and Tcp_notsent_lowat. https://blog.cloudflare.com/http-2-prioritization-with-nginx/.

[19] Optimum BTS. https://www.optimum.net/pages/speedtest.html.

[20] Source of Android / NetworkCapabilities.java. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/packages/Connectivity/framework/src/android/net/NetworkCapabilities.java.

[21] SpaceX Starlink Speeds Revealed as Beta Users Get Downloads of 11 to 60Mbps. https://arstechnica.com/information-technology/2020/08/spacex-starlink-beta-tests-show-speeds-up-to-60mbps-latency-as-low-as-31ms/.

[22] Speakeasy. https://www.speakeasy.net/speedtest/.

[23] Spectrum BTS. https://www.spectrum.com/internet/speedtest-only/.

[24] Speedof.me BTS. https://www.speedof.me/.

[25] SpeedTest BTS. https://www.speedtest.net.

[26] Speedtest.xyz BTS. https://speedtest.xyz/.

[27] TCP BBR Congestion Control Comes to GCP – Your Internet Just Got Faster. https://cloud.google.com/blog/products/gcp/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster.

[28] ThinkBroadband BTS. https://www.thinkbroadband.com/speedtest/.

[29] Understanding Internet Speeds. https://www.att.com/support/article/u-verse-high-speed-internet/KM1010095.

[30] Verizon BTS. https://www.verizon.com/speedtest/.

[31] WiFiMaster. https://en.wifi.com/wifimaster/.

[32] Xfinity BTS. http://speedtest.xfinity.com/.

[33] E. Alimpertis, A. Markopoulou, and U. Irvine. A System for Crowdsourcing Passive Mobile Network Measurements. In *Proc. of NSDI (2017)*. USENIX.

[34] V. Arun and H. Balakrishnan. Copa: Practical Delay-based Congestion Control for the Internet. In *Proc. of NSDI (2018)*, pages 329–342. USENIX.

[35] E. Atxutegi, F. Liberal, H. K. Haile, et al. On the Use of TCP BBR in Cellular Networks. *IEEE Communications Magazine (2018)*, 56(3):172–179.

[36] V. Bajpai and J. Schönwälder. A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts. *IEEE Communications Surveys & Tutorials (2015)*, 17(3):1313–1341.

[37] S. Bauer, D. Clark, and W. Lehr. Understanding Broadband Speed Measurements. *MIT Computer Science & Artificial Intelligence Lab, Tech. Rep. (2010)*.

[38] S. Bauer et al. Improving the Measurement and Analysis of Gigabit Broadband Networks. *SSRN 2757050 (2016)*.

[39] Z. S. Bischof, J. S. Otto, M. A. Sánchez, et al. Crowdsourcing isp characterization to the network edge. In *Proc. of SIGCOMM W-MUST workshop (2011)*, pages 61–66. ACM.

[40] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. of SIGCOMM (1994)*, pages 24–35. ACM.

[41] I. Canadi, P. Barford, and J. Sommers. Revisiting Broadband Performance. In *Proc. of IMC (2012)*, pages 273–286. ACM.

[42] N. Cardwell, Y. Cheng, C. S. Gunn, et al. BBR: Congestion-based Congestion Control. *Communications of the ACM (2017)*, 60(2):58–66.

[43] G. Casella, C. P. Robert, M. T. Wells, et al. Generalized Accept-reject Sampling Schemes. *A Festschrift for Herman Rubin (2004)*, pages 342–347.

[44] L. Chen et al. Auto: Scaling Deep Reinforcement Learning for Datacenter-scale Automatic Traffic Optimization. In *Proc. of SIGCOMM (2018)*, pages 191–205. USENIX.

[45] H. Dai, M. Shahzad, A. X. Liu, et al. Finding Persistent Items in Data Streams. In *Proc. of VLDB (2016)*, pages 289–300. VLDB Endowment.

[46] H. Deng, C. Peng, A. Fida, et al. Mobility Support in Cellular Networks: A Measurement Study on Its Configurations and Implications. In *Proc. of IMC (2018)*, pages 147–160. ACM.

[47] M. Dischinger, A. Haeberlen, K. P. Gummadi, et al. Characterizing Residential Broadband Networks. In *Proc. of IMC (2007)*, pages 43–56. ACM.

[48] M. Dong, Q. Li, D. Zarchy, et al. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proc. of NSDI (2015)*, pages 395–408. USENIX.

[49] M. Dong, T. Meng, D. Zarchy, et al. PCC Vivace: Online-Learning Congestion Control. In *Proc. of NSDI (2018)*, pages 343–356. USENIX.

[50] O. Goga et al. Speed Measurements of Residential Internet Access. In *Proc. of PAM (2012)*, pages 168–178. Springer.

[51] H. Goldstein, C. Poole, and J. Safko. *Classical mechanics*. American Association of Physics Teachers, 2002.

[52] S. Ha et al. CUBIC: a New TCP-friendly High-speed TCP Variant. In *Proc. of SIGOPS (2008)*, pages 64–74. ACM.

[53] N. Hu, L. Li, Z. M. Mao, et al. A Measurement Study of Internet Bottlenecks. In *Proc. of INFOCOM (2005)*, pages 1689–1700. IEEE.

[54] N. Hu, L. E. Li, Z. Mao, et al. Locating Internet Bottlenecks: Algorithms, Measurements, and Implications. In *Proc. of SIGCOMM (2004)*, pages 41–54. ACM.

[55] N. Hu and P. Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications (2003)*, 21(6):879–894.

[56] J. Huang et al. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *Proc. of SIGCOMM (2013)*, pages 363–374. ACM.

[57] A. Ivanov. Evaluating BBRv2 on the Dropbox Edge Network. *arXiv preprint arXiv:2008.07699 (2020)*.

[58] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *Proc. of INFOCOM (2004)*, pages 2490–2501. IEEE.

[59] A. Lehman. *JMP for Basic Univariate and Multivariate Statistics: a Step-by-step Guide*. SAS Institute, 2005.

[60] M. Levandowsky and D. Winter. Distance Between Sets. *Nature*, 234(5323):34–35, 1971.

[61] F. Li, A. A. Niaki, D. Choffnes, et al. A Large-scale Analysis of Deployed Traffic Differentiation Practices. In *Proc. of SIGCOMM (2019)*, pages 130–144. ACM.

[62] Y. Li, H. Deng, C. Peng, et al. icellular: Device-customized Cellular Network Access on Commodity Smartphones. In *Proc. of NSDI (2016)*, pages 643–656. USENIX.

[63] Y. Li et al. HPCC: High Precision Congestion Control. In *Proc. of SIGCOMM (2019)*, pages 44–58. ACM.

[64] B. Melander, M. Bjorkman, and P. Gunningberg. Regression-based Available Bandwidth Measurements. In *Proc. of International Symposium on Performance Evaluation of Computer & Telecommunication Systems Conference (SPECTS) (2002)*, pages 14–19. IEEE.

[65] B. Melander et al. A New End-to-End Probing and Analysis Method for Estimating Bandwidth Bottlenecks. In *Proc. of GlobeCom (2000)*, pages 415–420. IEEE.

[66] R. Mittal, V. T. Lam, N. Dukkipati, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. of SIGCOMM (2015)*, pages 537–550. ACM.

[67] A. Narayanan, J. Carpenter, E. Ramadan, et al. A First Measurement Study of Commercial mmWave 5G Performance on Smartphones. *arXiv preprint arXiv:1909.07532 (2019)*.

[68] J. Navratil and R. L. Cottrell. ABwE: A Practical Approach to Available Bandwidth Estimation. In *Proc. of PAM (2003)*, pages 14–19. Springer.

[69] T. Oshiba. Accurate Available Bandwidth Estimation Robust against Traffic Differentiation in Operational MVNO Networks. In *Proc. of ISCC (2018)*, pages 694–700. IEEE.

[70] V. Ribeiro, R. Riedi, R. Baraniuk, et al. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proc. of PAM workshop (2003)*. Springer.

[71] N. Sato, T. Oshiba, K. Nogami, et al. Experimental Comparison of Machine Learning-based Available Bandwidth Estimation Methods over Operational LTE Networks. In *Proc. of ISCC (2017)*, pages 339–346. IEEE.

[72] P. Schmitt et al. A Study of MVNO Data Paths and Performance. In *Proc. of PAM (2016)*, pages 83–94. Springer.

[73] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro area Mobile Connections. In *Proc. of IMC (2012)*, pages 301–314. ACM.

[74] L. Song and A. Striegel. Leveraging Frame Aggregation for Estimating Wifi Available Bandwidth. In *Proc. of SECON (2017)*, pages 1–9. IEEE.

[75] S. Sundaresan, X. Deng, Y. Feng, et al. Challenges in Inferring Internet Congestion using Throughput Measurements. In *Proc. of IMC (2017)*, pages 43–56. ACM.

[76] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-distance Networks. In *Proc. of INFOCOM (2004)*, pages 2514–2524. IEEE.

[77] T. Yang, Y. Jin, Y. Chen, et al. RT-WABest: A Novel End-to-end Bandwidth Estimation Tool in IEEE 802.11 Wireless Network. *International Journal of Distributed Sensor Networks (2017)*, 13(2):1550147717694889.

[78] F. Zarinni, A. Chakraborty, V. Sekar, et al. A First Look at Performance in Mobile Virtual Network Operators. In *Proc. of IMC (2014)*, pages 165–172. ACM.

[79] Y. Zhu, H. Eran, D. Firestone, et al. Congestion Control for Large-scale RDMA Deployments. In *Proc. of SIGCOMM (2015)*, pages 523–536. ACM.