12
Client-Side Experiments

The difference between theory and practice is larger in practice than the difference between theory and practice in theory
– Jan L.A. van de Snepscheut

Why you care: *You can run experiments either on a thin client, such as a web browser, or on a thick client, such as a native mobile app or a desktop client app. Changes for a webpage, regardless of whether it is frontend or backend, are fully controlled by the server. This is very different from a thick client. With an explosive growth of mobile usage, the number of experiments running on mobile apps has also grown* (Xu and Chen 2016). *Understanding the differences between thin and thick clients due to release process, infrastructure, and user behavior is useful to ensure trustworthy experiments.*

For most of this book, we assume thin clients when designing and running experiments to keep the discussions simple. This chapter is devoted to discussing running experiments in thick clients, their differences and implications.

Differences between Server and Client Side

To simplify the terminology, we will use "client-side experiment" to refer to experiment changes made within a thick client. We will use "server-side experiment" to refer to experiment changes made server side, regardless of whether it impacts a thick or thin client, and regardless of whether it is a UX change or a backend change.

There are two main differences between server and client side that impact online experiments: the release process and the data communication.

Difference #1: Release Process

On an online website, it is common for new feature releases to happen continuously, sometimes multiple times per day. Because changes are controlled by the organization, updating server-side code is relatively easy as part of continuous integration and deployment. When a user visits a site, the server pushes the data (e.g., HTML) to the browser, without interrupting the end-user experience. In a controlled experiment, the variant that the user sees is fully managed by the server and no end user action is required. Whether to show a red or yellow button, whether to show a newly revamped homepage or not—these are all changes that can happen instantaneously after a server-side deployment.

When it comes to client apps, many features can still be affected by services, that is, code on the server side, such as the feed content shown in the Facebook app. Changes affecting them would follow a similar release process as described above for a webpage. In fact, the more we can rely on services, the easier it is to experiment, both with regards to agility and consistency across different clients. For example, many changes on Bing, Google, LinkedIn, and Office are made server side and impact all clients, whether web or thick clients like mobile apps.

However, there is a significant amount of code shipped with the client itself. Any changes to this code must be released differently. For example, in a mobile app, developers do not have full control over the deployment and release cycle. The release process involves three parties: the app owner (e.g., Facebook), the app store (e.g., Google Play or Apple App Store), and the end user.

When the code is ready, the app owner needs to submit a build to the app store for review. Assuming the build passes review (which can take days), releasing it to everyone does not mean that everyone who visits the app will have the new version. Instead, getting the new version is a software upgrade, and users can choose to delay or even ignore the upgrade while continuing to use the old version. Some end users take weeks to adopt. Some enterprise organizations may not want updates and do not allow them for their users.

Some software, such as Exchange, runs in sovereign clouds that are restricted from calling unapproved services. All of these considerations mean that, at any given time, there are multiple versions of the app out there that the app owner has to support. Similar challenges exist for native desktop clients that have their own release mechanisms (e.g., Office, Adobe Acrobat, iTunes), even though there may not be an app store review process involved.

It is worth pointing out that both the Google Play and Apple app stores now support staged rollout (Apple, Inc. 2017, Google Console 2019). They both allow app owners to make the new app available to only a percentage of users and pause if something problematic is discovered. Staged rollouts are essentially randomized experiments, as the eligible users are selected at random. Unfortunately, these rollouts cannot be analyzed as random experiments because the app owners do not know which users are eligible to receive the new app. App owners only know who has "adopted" the new app. We will discuss this more later in this chapter.

App owners may not want to frequently push a new client version. Even though there is no strict limit as how many times they can release a new version, each update costs network bandwidth for users and can potentially be an annoying user experience (depending on update and notification settings). Windows or iOS is a great example of something that cannot update as often because some updates require a reboot.

Difference #2: Data Communication between Client and Server

Now that the new app is in the hands of users, it has to communicate with the server. The client needs to get the necessary data from the server, and it needs to pass data back to the server on what is happening on the client. While we refer readers to Chapter 13 for client-side instrumentation in general, here we highlight some key factors when it comes to data communication for a native mobile app. While it makes it easier to read this section with mobile in mind, please note that with drastic technology improvement, the divide between mobile and desktop is becoming nominal as a reflection of device capabilities and improvement in network connections.

First, the data connection between the client and server may be limited or delayed:

- **Internet connectivity**. Internet connections may not be reliable or consistent. In some countries, users may be offline for days. Even users who are normally online may not have internet access on a plane or be temporarily in a zone with no available cellular or Wi-Fi networks. As a result, data changes happening server side may not be pushed to these clients. Similarly, data collection on the client may be delayed in transmitting back to the server. These delays vary by country or demographic and must be accounted for in instrumentation and downstream processing.
- **Cellular data bandwidth**. Most users have limited cellular data plans, which raises the question of whether you only upload telemetry when the user is on Wi-Fi or at any point in time. Most apps choose to send telemetry

data over Wi-Fi only, which can delay when that data is received server side. There can also be heterogeneity across countries, as mobile infrastructure in some countries is weaker than others when it comes to bandwidth, cost, and so on.

Not only could the data connection itself be limited, but even if the connection is good, using the network may impact device performance and ultimately user engagement with the app (Dutta and Vadermeer 2018):

- **Battery**. More data communication implies increased battery consumption. For example, the app can wake up more regularly to send more telemetry, but that would impact battery consumption. Moreover, mobile devices in low battery mode have restrictions on what apps are allowed to do (Apple, Inc. 2018).
- **CPU, latency, and performance**. Even though many mobile devices behave like minicomputers nowadays, there are still lower-end mobile devices constrained by CPU power. Frequent data aggregation on the device and sending data back-and-forth with the server can make the app less responsive and hurt the overall performance of the app.
- **Memory and storage**. Caching is one way to reduce data communication but impacts the size of the app, which impacts app performance and increases app uninstallment (Reinhardt 2016). This may be a larger concern for users with lower-end devices with less memory and storage.

Communication bandwidth and device performance are all part of the same device ecosystem, with tradeoffs. For instance, we can get a more consistent internet connection by using more cellular data; we can spend more CPU to compute and aggregate on-device to reduce data sent back to the server; we can wait for Wi-Fi to send tracking data by using more storage on-device. These tradeoffs can impact both visibility into what is happening client side, as well as user engagement and behavior (similar to Chapter 5), making it a fruitful area for experimentation but also an area where care needs to be taken to ensure trustworthy results.

Implications for Experiments

Implication #1: Anticipate Changes Early and Parameterize

As client code cannot be shipped to end users easily, controlled experiments on any client-side changes need to be planned. In other words, all experiments, including all variants for each of these experiments, need to be coded and shipped with the current app build. Any new variants, including bug fixes on any existing variant, must wait for the next release. For example, in a typical monthly release, Microsoft Office ships with hundreds of features that rollout in a controlled manner to ensure safe deployment. This has three implications:

1. A new app may be released before certain features are completed, in which case, these features are gated by configuration parameters, called feature flags, that turn the features off by default. Features turned off this way are called dark features. When the feature is finished and ready, sometimes when the server-side service completes, it can be turned on.
2. More features are built so they are configurable from the server side. This allows them to be evaluated in A/B tests, which helps both in measuring performance via controlled experiments, and also provides a safety net. If a feature does not perform well, we can instantly revert by shutting down the feature (the variant in the controlled experiment) without having to go through a lengthy client release cycle. This can prevent end users from being stuck with a faulty app for weeks until the next release.
3. More fine-grained parameterization can be used extensively to add flexibility in creating new variants without needing a client release. This is because even though new code cannot be pushed to the client easily, new configurations can be passed, which effectively creates a new variant if the client understands how to parse the configurations. For instance, we may want to experiment on the number of feed items to fetch from the server at a time. We could put our best guess in the client code and only experiment with what we planned, or we can parameterize the number and have the freedom to experiment post release. Windows 10 parameterized the search box text in the task bar, ran experiments over a year after it shipped, with the winning variant increasing user engagement and Bing revenue by millions of dollars. Another common example is to update machine learning model parameters from the server, so that a model can be tuned over time.

While we believe it is best for the user experience to test new features before launching to all app users, there may be limitations imposed by app stores on which features can be shipped dark. We suggest carefully reading app store policies and appropriately disclosing dark features.

Implication #2: Expect a Delayed Logging and Effective Starting Time

The limited or delayed data communication between client and server cannot only delay the arrival of data instrumentation, but also the start time of the experiment itself. First, the experiment implementation on the client side needs to be shipped with the new app version. Then, we can activate the experiment for a small percentage of users. However, even then, the experiment is not fully active because:

- User devices may not get the new experiment configuration, either because devices are offline or because they are in limited or low bandwidth situations, where pushing new configurations can lead to increased costs or poor experiences for the user.
- If the new experiment configuration is fetched only when a user opens the app, the new assignment may not take effect until the next session as we do not want to change a user's experience after they have started the current session. For heavy users with multiple sessions a day, this delay is small, but for light users visiting once a week, the experiment may not start until a week later.
- There can be many devices with old versions without the new experiment code, particularly right after the new app release. Based on our experience, the initial adoption phase takes about a week to reach a more stable adoption rate, although this can vary greatly depending on the user population and the type of app.

These delays in experiment start time and instrumentation arriving on the server can impact experiment analysis, especially if the analysis is time sensitive, for example, real-time or near real-time. First, signals at the beginning of the experiment would appear to be weaker (smaller sample size), and also have a strong selection bias towards frequent users and Wi-Fi users who tend to be early adopters. Thus, the duration of an experiment may need to be extended to account for delays. Another important implication is that Treatment and Control variants may have a different effective starting times. Some experiment platforms allow for shared Control variants, in which case the Control variant may be live before the Treatment, and therefore have a different user population due to selection bias. In addition, if the Control runs earlier, the caches are warmed up so responses to service requests are faster, which may introduce additional bias. As a result, the time period to compare the Treatment and Control needs to be carefully chosen.

Implication #3: Create a Failsafe to Handle Offline or Startup Cases

When users open an app, their device could be offline. For consistency reasons, we should cache experiment assignment in case the next open occurs when the device is offline. In addition, if the server is not responding with the configuration needed to decide on assignment, we should have a default variant for an experiment. Some apps are also distributed as original equipment manufacture (OEM) agreements. In these cases, experiments must be properly set up for a first-run experience. This includes retrieving configurations that would only impact the next startup, and a stable randomization ID before and after users sign up or log in.

Implication #4: Triggered Analysis May Need Client-Side Experiment Assignment Tracking

You may need to take additional care to enable triggered analysis for client-side experiments. For example, one way to capture triggering information is to send tracking data to the server when an experiment is used. However, to reduce communications from the client to the server, experiment assignment information is usually fetched for all active experiments at once (e.g., at the start of the app), regardless of whether an experiment is triggered or not (see Chapter 20). Relying on the tracking data at fetching time for triggered analysis would lead to over-triggering. One way to address this problem is to send the assignment information when a feature is actually used, thus requiring experiment instrumentation to be sent from the client. Keep in mind that if the volume of these tracking events is high, it could cause latency and performance issues.

Implication #5: Track Important Guardrails on Device and App Level Health

Device-level performance may impact how the app performs. For instance, the Treatment may be consuming more CPU and draining more battery power. If we only track user engagement data, we may not discover the battery-drain problem. Another example is that the Treatment may send more push notifications to users that then lead to an increased level of notification disablement via device settings. These may not show up as a significant engagement drop during the experiment but have a sizable long-term impact.

It is also important to track the overall health of the app. For example, we should track app size, as a bigger app size is more likely to reduce downloads and cause people to uninstall (Tolomei 2017, Google Developers 2019). Similar behaviors may result due to an app's internet bandwidth consumption, battery usage, or crash rate. For crashes, logging a clean exit allows sending telemetry on a crash on the next app start.

Implication #6: Monitor Overall App Release through Quasi-experimental Methods

Not all changes on the new app can be put behind an A/B parameter. To truly run a randomized controlled experiment on the new app as a whole, bundle both versions behind the same app and start some users on the new version, while keeping others on the old version. This is not practical or ideal for most apps, as it can double the app size. On the other hand, because not all users adopt the new app version at the same time, there is a period of time where we have both versions of the app serving real users. This effectively offers an A/B comparison if we can correct for the adoption bias. Xu and Chen (2016) share techniques to remove the bias in the mobile adoption setting.

Implication #7: Watch Out for Multiple Devices/Platforms and Interactions between Them

It is common for a user to access the same site via multiple devices and platforms, for example, desktop, mobile app, and mobile web. This can have two implications.

1. Different IDs may be available on different devices. As a result, the same user may be randomized into different variants on different devices. (Dmitriev et al. 2016).
2. There can be potential interactions between different devices. Many browsers, including Edge, now have a "Continue on desktop" or "Continue on mobile" sync feature to make it easier for users to switch between desktop and mobile. It is also common to shift traffic between the mobile app and mobile web. For example, if a user reads an e-mail from Amazon on their phone and clicks it, the e-mail link can either take them directly to the Amazon app (assuming they have the app installed) or to the mobile website. When analyzing an experiment, it is important to know whether it may cause or suffer from these interactions. If so, we cannot evaluate app performance in isolation, but need to look at user behavior holistically across different platforms. Another thing to watch for is that the user experience on one platform (usually the app) can be better than on another platform. Directing traffic from the app to the web tends to bring down total engagement, which may be a confounding effect not intended by the experiment itself.

Conclusions

We have devoted this chapter to the differences when experimenting on thin vs. thick clients. While some differences are obvious, many are subtle but critical. We need to put in extra care in order to design and analyze the experiments properly. It is also important to point out that with rapid technological improvement, we expect many of the differences and implications to evolve over time.