Web Performance Calendar

The speed geek's favorite time of year
2020 Edition

Strategies for Telemetry Exfiltration (aka Beaconing In Practice)

28thDec 2020 by Nic Jansma
ABOUT THE AUTHOR

Nic Jansma (@nicj) is a software developer at Akamai building high-performance websites, apps and open-source tools.

Table of Contents

Introduction

- Step 1: Gather the data!
- Step 2: ???
- Step 3: Profit!

Let's say you have a website, and you want to find out how long it takes your visitors to see the Largest Contentful Paint on your homepage.

Or, let's say you want to track how frequently your visitors are clicking a button during the Checkout process.

Or, let's say you want to use the new Measure Memory API to track JavaScript memory usage over time, because you're concerned that your Single Page App might have a leak.

Or, let's say your work on a performance analytics library that automatically captures performance metrics all throughout the Page Load and beyond.

For each of those scenarios, you may end up using one of the many exciting JavaScript APIs or libraries to capture, query, track or observe key metrics.

That's the easy part!

The hard part is making sure your back-end actually **receives** that data in a **reliable** way. If your telemetry hasn't been received, the experience never happened! What's worse, **you may not even know that you don't know** it happened!

So, I'd argue that Step 2 is just as important as Step 1:

- Step 1: Gather the data!
- Step 2: **Beacon the data!**
- Step 3: Profit!

This article will look at several strategies for reliably exfiltrating telemetry — aka **beaconing**. We will cover **when** and **how** to send beacons, and gotchas you should watch out for.

This article was written by one of the authors of Boomerang, an open-source RUM performance monitoring library that sends a *lot* of beacons (1 billion+ a day!). We were taking a look at *how* and *when* we send beacons to make sure we're sending them as optimally as possible, especially to make sure we're not missing beacons due to listening to the wrong (or too many) events. See our findings in the TL;DR section!

Beacons

Each of the scenarios above cover different ways that websites can collect **telemetry**. What is telemetry? Wikipedia says:

> Telemetry is the in situ collection of measurements or other data at remote points and their **automatic transmission** to receiving equipment (telecommunication) for monitoring

Any sort of measurement, whether it's for performance, marketing or just curiosity, is telemetry data. We generally collect telemetry to improve our websites, our services and our visitor's experiences.

Your website may have its own internal telemetry that tracks application health, or you may rely on third-party marketing or performance analytics libraries to collect data for you automatically.

An essential part of **collecting** telemetry is making sure that it is reliably **sent** (exfiltrated) so you can actually **use** it (in bulk).

In analytics terms, we often call sending telemetry **beaconing**, and the HTTPS payload that carries the data the **beacon**.

Beaconing Stages

Every time you collect some data, you should have a strategy for when you're going to get that data out of the browser.

This sounds simple, but depending on the type of data you're tracking, *when* you send it matters just as much as collecting it.

Let's look at some common scenarios:

Sending Data at Startup

Sometimes, you just want to log that a thing happened. For example, you can log when a Page Load occurred and maybe include a few extra bits of details, like the URL that was loaded or characteristics of the browser.

As long as you're not waiting on anything else, in this case, it makes sense to **beacon immediately** after the analytics code has loaded.

Many marketing analytics scripts, such as Google or Adobe Analytics fall into this bucket. As soon as their JavaScript libraries are loaded, they may immediately send a beacon noting that "this Page Load happened" with supporting details about the Page Load's dimensions.

```
// pseudo code
function onStartup() {
  // gather the data
  sendBeacon();
}
```

Good for:

- Quick marketing-level analytics
- Highly reliable

Bad for:

- Collecting any Page Load **performance** data

- Measuring anything that happens after the page has loaded (e.g. user interactions or post-Load content)

Gathering Data through the Page Load

Some websites use Real User Monitoring (RUM) to track the performance of each Page Load. Since you're waiting for the Page Load to finish, you can't immediately send a beacon when the JavaScript starts up. Generally, you'll need to wait for at least the Page Load (`onload`) event, and possibly longer if you have a Single Page App.

To do so, you would normally register for an `onload` handler, then send your data immediately after the `onload` event has finished.

Performance analytics libraries such as boomerang.js or SpeedCurve's LUX will wait until the Page Load (or SPA Page Load) events before beaconing their data.

```
// pseudo code
function onStartup() {
  window.addEventListener('load', function(event) {
    // you may want to capture more data now, such as the total Page Load time
    gatherMoreData();

    sendBeacon();
  });

  // you could collect some details now, such as the page URL
  gatherSomeData();
}
```

Note: You may want to delay your beacon until slightly *after* `onload` to ensure your analytics tool doesn't cause a lot of work at the same time other `onload` handlers are executing:

```
// pseudo code
function onStartup() {
  window.addEventListener('load', function(event) {
    // wait a little bit until Page Load activity dies down
    setTimeout(function() {
      // you may want to capture more data now, such as the total Page Load time
      gatherMoreData();

      sendBeacon();
    }, 500);
  });

  // you could collect some details now, such as the page URL
  gatherSomeData();

  // ALSO!  Have an unload strategy
}
```

Good for:

- Gathering performance analytics

Bad for:

- Measuring anything that happens after the page has loaded (e.g. user interactions or post-Load content)
- Waiting *only* for the Page Load event means you will miss data from any user that abandons the page *prior* to Page Load
- Make sure you have an *unload strategy* to capture abandons.

Incrementally Gathering Telemetry throughout a Page's Lifetime

After the page has loaded, there may be user interactions or other periodic changes to the page that you want to track.

For example, you may want to measure how many times a button is clicked, or how long it takes for that button click to result in a UI change.

This type of on-the-fly data collection can often be exfiltrated immediately, especially if you’re tracking events in real-time:

```
// pseudo code
myButton.addEventListener('click', function(event) {
  sendBeacon();
});
```

You could also consider *batching* these types of events and sending the data periodically. This may save a bit of CPU and network activity:

```
// pseudo code
var dataBuffer = [];
myButton.addEventListener('click', function(event) {
  dataBuffer.push(...);
});

// send every 10 seconds if there's new data
setInterval(function() {
  if (dataBuffer.length) {
    sendBeacon(dataBuffer);
    dataBuffer = [];
  }
}, 10000);
```

Good for:

- Real time event tracking

Bad for:

- If you’re batching data, you should have an *unload strategy* to ensure it goes out before the user leaves

Gathering Data up to the End of the Page

Some types of metrics are continuous, happening or updating throughout the page’s lifecycle. You don’t necessarily want to send a beacon for every update to those metrics — you just want to know the “final” result.

One simple example of this is when measuring Page View Duration, i.e. how long the user spent reading or viewing the page. Sure, you could send a beacon every minute (“they’ve been viewing for [n] minutes!”), but it’s a lot more efficient to just send the final value (“they were here for 5 minutes!”) **once**, when the user is navigating away.

If you’re interested in Google’s Core Web Vitals metrics, you should probably track Cumulative Layout Shift (CLS) beyond just the Page Load event. If Layout Shifts happen post-page-load, those also affect the user experience. CLS is a score that incrementally updates with each Layout Shift, so you shouldn’t necessarily beacon on each Layout Shift — you just want the final CLS value, after the user leaves the page.

Another example would be for the Measure Memory API, which lets you track memory usage over time. If your Single Page App is alive for 3 hours (over many interactions), you may only want to send one final beacon with how the memory behaved over that lifetime.

For these cases, your best bet is to listen for a page lifecycle indicator like the `pagehide` event, and send data as the user is navigating away. The specific events you want to listen for are a little complex, so read up on unload strategies later.

```
// pseudo code
var clsScore = 0;

// don't listen for just pagehide!  see unload stragies section
window.addEventListener('pagehide', function(event) {
  sendBeacon();
});

// Listen for each Layout Shift
var po = new PerformanceObserver(function(list) {
  var entries = list.getEntries();
  for (var i = 0; i < entries.length; i++) {
    if (!entries[i].hadRecentInput) {
      clsScore += entries[i].value;
    }
  }
});

po.observe({type: 'layout-shift', buffered: true});
```

Good for:

- Continuous metrics that are updated over time, and you only want the final value

Bad for:

- Real time metrics — these will be delayed until the user actually navigates away
- Reliability — you will lose some of this data just because unload events aren't as reliable, so have an unload strategy

"Whenever"

Sometimes you may want track metrics or events, but you don't necessarily need to send the data immediately (because it doesn't need to be *Real Time* data). In fact, it may be advantageous to delay sending until *another beacon* has to go out. For example, as a later beacon is flushed, you can tack on additional data as needed.

In this case, you may want to:

- Send data on the next outgoing beacon, if any
- Send batched data periodically, if desired
- Send any un-sent data at the end of the page

To do this, you would use a combination of the strategies above — using queuing/batching and unload beacons.

Good for:

- Minimizing beacon counts

Bad for:

- Real-time metrics
- Reliability — you will lose some of this data just because unload events aren't as reliable, so have an unload strategy

How Many Beacons?

Depending on the data you're collecting, and how you're considering exfiltrating it, you may have the choice to send a single beacon, or multiple beacons. Each has its own advantages and disadvantages, from the client's (browser's) perspective, as well as the server's.

A Single Beacon

A single beacon is the simplest way to send your data. Collect all of your data, and when you're done, send out a single beacon and stop processing. This is frequently how marketing and performance analytics beacons are implemented, when sending the results of a single Page Load.

Good for:

- Less processing (CPU) time in the client
- Less network egress bytes (less protocol overhead of a single network request vs. multiple requests)
- Easier on the back-end — all data relating to the user experience is in one beacon payload, so the server doesn't have to stitch it back together later

Bad for:

- Real-time metrics, unless you're sending the beacon early in the Page Load cycle (immediately or at `onload`).
- Capturing data after the beacon has been sent

Multiple Beacons

If you're collecting data at multiple stages throughout the page lifecycle, or due to user interactions, you may want to send that data on multiple beacons.

The main downside to multiple beacons is that it *costs more* from several perspectives: more JavaScript CPU time building the beacons, more network overhead sending the beacons, more server CPU time processing the beacons.

In addition, depending on how the back-end server infrastructure is setup, you may want to "link" or "stitch" those beacons together. For example, let's say you're interested in tracking the Load Time of a Page, as well as the final Cumulative Layout Shift Score. You may send a beacon out at the `onload` event with the Load Time, but wait until the `unload` event to send the final CLS Score.

Later, when you're analyzing the data, you may want to group or compare Page Load times with their final CLS Scores. To do that, you would need to link the beacons together through some sort of GUID, and probably spend time on the back-end *joining* those beacons together (at your database layer).

An alternative strategy, once the Page Load beacon arrives, is holding it in memory until the final CLS Score arrives, before "stitching" it together on the back-end and sending to the database as a "combined" beacon with all of the data of that Page Load Experience. Doing this would result in additional server complexity, memory usage, and probably less reliability. You'd also need to figure out what happens if one of the partial beacons never arrives (data gets lost in-transit all the time, and sometimes events like `unload` never fire).

If you'll never be looking at or comparing the data from those multiple beacons, these concerns may not matter. But if you're doing more advanced analytics where joining data from multiple beacons would be common, you should weigh the pros and cons of multiple beacons as part of your strategy.

Good for:

- Real-time capturing/reporting of events, events don't "wait" for a later beacon to be sent
- Capturing data beyond a single event, throughout a Page Load lifecycle

Bad for:

- Generally more processing time on the client (preparing the beacon)
- Generally more network usage (HTTP protocol overhead, repeated dimensions or IDs to stitch to other beacons)
- Generally more processing on the server (multiple incoming requests)
- Harder to keep context of the same user experience together — multiple beacons may need to be "joined" for querying or held in-memory until they all arrive

Mechanisms

Once you've figured out when you'd like to send your beacon(s), and how many you'll send, you need to convince the browser to send it. There's at least 4 common APIs to send beacons: `Image`, `XMLHttpRequest`, `sendBeacon()` and Fetch API.

Image

The simplest method of beaconing data is by using a HTML `Image`, commonly called a “pixel”. This is generally done via a HTTP GET request by creating a hidden DOM `Image`, setting its `Image.url`, and including your beacon data in the query string.

Often, the server will respond with a `204 No Content` or a simple/transparent 1×1 pixel image.

```
var img = new Image();
img.src = 'https://site.com/beacon/?a=1&b=2';
```

You can’t include any data in the “body” of the `Image`, as you only have the URL (query string) to work with. This limits you to how much actual data can be sent, depending on both the browser and server configuration.

From the browser’s point of view, most modern browsers support URL lengths of at least 64 KB:

- Chrome: ~ 100 KB
- Firefox (3.x): >= 5 MB
- Firefox (recent): ~ 100 KB
- Safari 4, 5: >= 5 MB
- Safari 13: ~ 64 KB
- Mobile Safari 13: ~ 64 KB
- Internet Explorer 6, 7: 2083 bytes
- Internet Explorer 8, 9, 10, 11: >= 5 MB
- Edge (EdgeHTML 20-44): >= 5 MB
- Edge (Chromium 79+): ~ 100 KB
- Opera (Presto <= 12): >= 5 MB
- Opera (Chromium): ~ 100 KB

Notably small exceptions are Internet Explorer 6 and 7 (… does anyone still care?).

One thing to keep in mind is that serializing data onto the URL is usually inefficient. Strings need to be URI-encoded, which bloats the size of characters due to “percent encoding”. Especially if you’re trying to tack on raw JSON, like this:

```
{"abc":123,"def":"ghi"}
```

It gets expanded on the URL by 69% to:

```
%7B%22abc%22:123,%22def%22:%22ghi%22%7D
```

You may be able to minimize this type of bloat by using compression or things like JSURL.

The browser’s URL limits are just part of the story. Most web servers also have their own max request URL size:

- Apache: Defaults to 8190 bytes and can be increased via the `LimitRequestLine` directive
- TomCat has a default limit of 8 KB, and can be increased up to 64 KB via `maxHttpHeaderSize`
- Jetty has a default limit of 8 KB, and can be increased via `requestHeaderSize`
- CDNs will have their own URL length limits, which are usually not configurable. Akamai, CloudFront and Fastly all seem to have limits around 8KB.
- Users may have proxies installed that have their own limits

At the end of the day, it’s safest to limit `Image` beacon URLs to under 2,000 bytes, if you care about Internet Explorer 6 and 7. If not, you can probably go up to 8,190 bytes unless you’ve specifically configured and tested all of the parts of your CDN and server infrastructure.

I’m not specifically aware of any user proxies with URL limits, but my guess is there are some out there that may have limits around the same sizes (of 2 or 8 KB), so even if your server infrastructure supports longer request URLs, *some* users may not be able to send requests that long.

`Image` Beacon Pros:

- Simplest API

- Least amount of overhead
- Largest browser support
- Will not be rejected or delayed by CORS

`Image` Beacon Cons:

- Does not support HTTP POST
- Does not support any payload other than the URL
- Does not support more than ~2 KB of data, depending on the browser
- Not as reliable as `sendBeacon()`

XMLHttpRequest

Once the `XMLHttpRequest` (XHR) API was added to browsers, it created a way for developers to use the API to send raw data to any URL, instead of pretending we were fetching `Images` from everywhere.

XHRs are a lot more flexible than `Image` beacons. They can use any HTTP method, including POST. They can also include a body payload (of any `Content-Type`), so we can avoid the URL length concerns of `Image` beacons.

To avoid the CORS performance penalty of a `OPTIONS` Pre-Flight, you should make sure your XHR beacon is a simple request: only GET/POST/HEAD, no fancy headers, and a `Content-Type` of either:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

Make sure to review the fallback strategies in case `XMLHttpRequest` isn't available, or if it fails.

XHR allows you to send data *synchronously* or *asynchronously*. There's really no reason to send synchronous XHRs these days. Some websites used to send synchronous XHRs on `unload` to make sure the beacon data was sent prior to the browser closing the page. These days, you should use `sendBeacon()` instead for even more reliability and better performance.

Here's an example of using XHR to send a beacon with multiple key-value pairs:

```
// data to send
var data = {
  a: 1,
  b: 2
};

// open a POST
var xhr = new XMLHttpRequest();
xhr.open('POST', 'https://site.com/beacon/');
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

// prepare to send our data as FORM encoded
var params = [];
for (var name in data) {
  if (data.hasOwnProperty(name)) {
    params.push(encodeURIComponent(name) + '=' + encodeURIComponent(data[name]));
  }
}

var paramsJoined = params.join('&');

// send!
xhr.send(paramsJoined);
```

`XMLHttpRequest` Beacon Pros:

- Simple API
- Supports HTTP POST and other methods
- Supports a payload in the body of any content type

- Supports any size payload (up to server limits)

`XMLHttpRequest` Beacon Cons:

- May require consideration around CORS to avoid Pre-Flights
- Not as reliable as `sendBeacon()`

sendBeacon

The `navigator.sendBeacon(url, payload)` API provides a mechanism to asynchronously send beacon data more performantly and reliably than using `XMLHttpRequest` or `Image`. When using the `sendBeacon()` API, even if the page is about to unload, the browser will make a best effort attempt to send the data. The request is always a HTTP POST.

`sendBeacon()` was built for telemetry, analytics and beaconing, and we should use it if available! According to caniuse.com, over 95% of browser marketshare supports `sendBeacon()` today (the end of 2020).

The API is fairly simple to use on its own, but has a few gotcha's and limits.

First, the return value of `navigator.sendBeacon()` should be checked. If it returned `true`, you've successfully handed data off to the browser and you're good to go! Note this doesn't mean the data *arrived* at the server — you'll never be able to see the server's response to the beacon with the `sendBeacon()` API.

The `sendBeacon()` API will return `false` if the UA could not queue the request. This generally happens if the payload size has tripped over certain beacon limits that the browser has set for the page. Here's what the Beacon API spec says about these limits:

> The user agent imposes limits on the amount of data that can be sent via this API: this helps ensure that such requests are delivered successfully and with minimal impact on other user and browser activity. If the amount of data to be queued exceeds the user agent limit, this method returns false; a return value of true implies the browser has queued the data for transfer. However, since the actual data transfer happens asynchronously, this method does not provide any information whether the data transfer has succeeded or not.

In practice today, the following limits are observed:

- Firefox does not appear to impose any limits
- Chromium-based browsers and Safari have:
    - A payload size limit: this is defined in the Fetch API spec as 64 KB
    - An outstanding-beacon payload limit: if there are other `navigator.sendBeacon()` requests in progress (from any script), and the sum of their payload sizes is over 64 KB, the limit is breached
- In Chrome versions earlier than 66, if the total size of **previous calls** to `sendBeacon()` was over 64 KB, subsequent calls would fail

Besides these limits, the URL itself *could* also contain data, and would adhere to the same URL limits seen in the `Image` beacon section.

If the `navigator.sendBeacon()` returns `false`, it means the browser *will not* be sending the beacon. If so, it's best to fallback to `XMLHttpRequest` or `Image` beacons.

This sample code will check that `sendBeacon()` exists and works, and if not, fallback to XHR/Image beacons:

```
function sendData(payload) {
  if (window &&
    window.navigator &&
    typeof window.navigator.sendBeacon === "function" &&
    typeof window.Blob === "function") {

    var blobData = new window.Blob([payload], {
      type: "application/x-www-form-urlencoded"
    });

    try {
      if (window.navigator.sendBeacon('https://site.com/beacon/', blobData)) {
        // sendBeacon was successful!
        return;
      }
    } catch (e) {
      // fallback below
    }
  }

  // Fallback to XHR or Image
  sendXhrOrImageBeacon();
}
```

Note there are only 3 CORS safelisted `Content-Types` you can send:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

Any other content type will result in a CORS pre-flight for cross-origin requests, which isn't desired for a beacon that you're trying to get out reliably. So if you're wanting to send `application/json` content to another domain, you may consider encoding it as just `text/plain`.

`sendBeacon` Pros:

- Simple API, but beware of fallbacks
- Most reliable
- Should not be rejected or delayed by CORS (using the correct `Content-Types`)
- Supports any size payload, though the browser may reject larger sizes (stick to under 64 KB)

`sendBeacon` Cons:

- Calling it does not guarantee the API will "accept" the call — you may need to fallback to other metrics
- Only supports HTTP POST
- Supports only some Content Types to avoid CORS pre-flight

Fetch API

Similar to using an `XMLHttpRequest`, the modern `fetch()` API could be used to send beacons. If you're already using Fetch in your app, you could use that interchangeably with `XMLHttpRequest` as a fallback.

In addition, there's a recent Fetch API option called `keepalive: true`. This option is likely what `sendBeacon()` is using under the hoods in most browsers.

This is supported by Chrome 66+, Safari 11+, and is being considered by Firefox.

There are some caveats and limitations around using `keepalive` so I'd encourage you to review that issue if you're using the Fetch API.

At this point, I'd suggest using `sendBeacon()` over the Fetch API.

Fallback Strategies

Not every beaconing method is available in every browser. You'll want to try to fallback to older methods if `sendBeacon()` isn't available:

Generally, use:

1. `sendBeacon()` if available (for reliability) and if it returns `true`
2. `XMLHttpRequest` (or Fetch API) if you need to use HTTP POST **or** have a body payload **or** if the data is > 2 KB
3. `Image` otherwise

Payload

What does your data look like? How big is it?

Ideally, you should minimize the outgoing request size as much as possible to avoid overtaxing your visitor's network. To do this, you could consider various forms of data minification or compression.

Limits

It would be wise to first look at your expected minimum, median and maximum payload size. This may dictate what kind of beacon you can send, i.e. `Image` vs `XMLHttpRequest` vs `sendBeacon()`, and whether any sort of minification/compression is needed.

Briefly:

- If your data is under 2 KB, you can use any type of beacon, and probably don't need to compress it
- If your data is under 8 KB, you can use any type of beacon, but won't support IE 6 or 7
- If your data is under 64 KB, you can use `sendBeacon()` or `XMLHttpRequest`, and you may want to consider compressing it
- If your data is over 64 KB, you can only use `XMLHttpRequest`, and you may want to consider compressing it

Payload via URL (Query String)

The simplest beacons can include all of their data in the Query String of a URL, i.e.:

```
https://mysite.com/beacon/?a=1&b=2...
```

As we saw with the `Image` beacon section, in practice this is limited to a total URL length of 2 KB (if you support IE 6/7) or 8 KB (unless your server infrastructure supports more).

One complication is that characters outside of the range below will need to be URI-encoded by `encodeURIComponent`:

```
A-Z a-z 0-9 - _ . ! ~ * ' ( )
```

Depending on your data, this could bloat the size of your URL significantly! You may want to consider JSURL or another compression technique to help offset this if you're sticking to a URL payload.

Payload via Request Body

For `XMLHttpRequest` and `sendBeacon` calls, you'll often specify the bulk of your data in the **payload** of the beacon (instead of the URL).

Commong ways of encoding your beacon data include:

- `multipart/form-data` via `FormData`, which is pretty inefficient for sending multiple small key-value pairs due to the "boundary" and `Content-Disposition` overhead:

  ```
  ------WebKitFormBoundaryeZAm2izbsZ6UAnS8
  Content-Disposition: form-data; name="a"

  1
  ------WebKitFormBoundaryeZAm2izbsZ6UAnS8
  Content-Disposition: form-data; name="b"

  2
  ------WebKitFormBoundaryeZAm2izbsZ6UAnS8--
  ```

- `application/x-www-form-urlencoded` (via `UrlSearchParams`), which suffers from the same percentage encoding bloat as URLs if you have many non-alpha-numeric characters.

- `text/plain` with whatever text content you want, if your server knows how to parse it

Any other content type may trigger a CORS pre-flight for cross-origin requests in `XMLHttpRequest` and `sendBeacon`.

Compression

You may want to consider reducing the size of your URL or Body payloads, if possible. There are always trade-offs in doing so, as minification/compression generally use CPU (JavaScript) to reduce outgoing byte sizes.

Some common techniques include:

- Using a data-specific compression technique to reduce or minify data. We have some examples for data compression in Boomerang for ResourceTiming and UserTiming.
- URL and `application/x-www-form-urlencoded` body payloads can benefit from being minified by JSURL, which swaps out characters that must be encoded for URL-safe characters.
- The Compression Streams API could be used to compress large payloads for browsers that support it

Reliability

As described above, there are many different stages of the page lifecycle that you can send data. Often, you'll want to send data during one of the lifecycle events like `onload` or `unload`.

Browsers give us a lot of lifecycle events to listen to, and depending on which of these events you use, you may be more-or-less likely to **receive** data if you send a beacon then.

Let's look at some examples, and find a strategy for when to send our beacons, so we can have the best reliability of the data reaching our servers.

Methodology

I recently ran a study on one of my websites, collecting data over a week from a large set (millions+) of Page Loads.

For each of these visitors, I sent multiple beacons: as soon as the page started up, at `onload`, during `unload` and several other events.

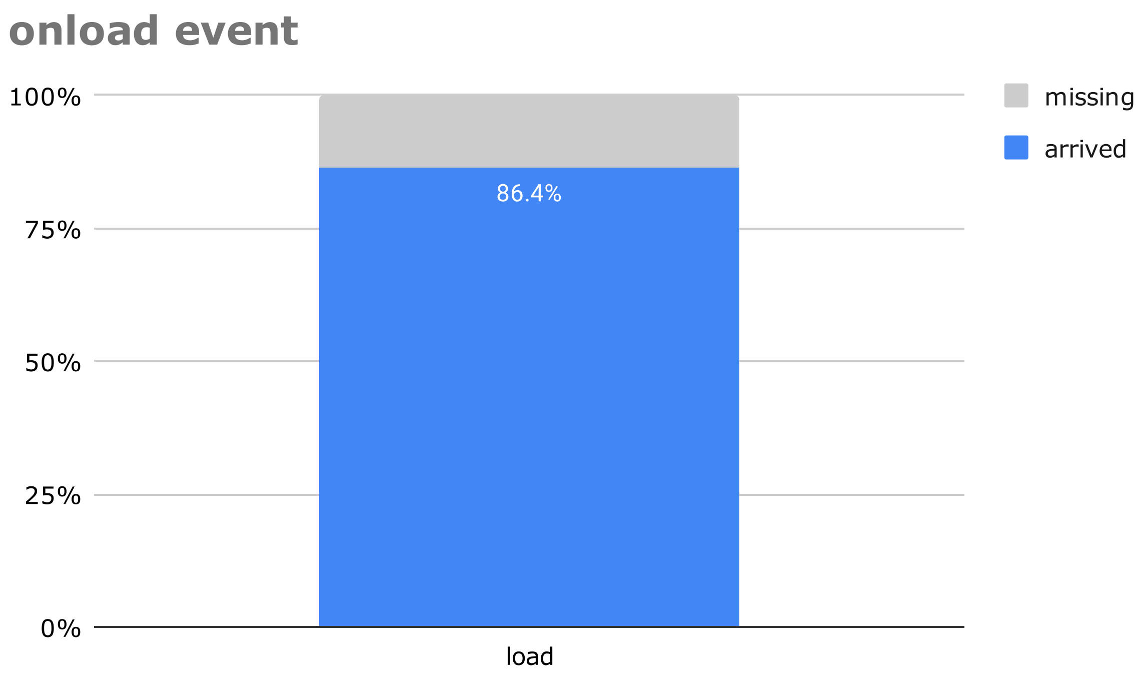
The goal was to see how reliable beaconing is at each of those events, and to see what combination of events would be the most reliable way to receive beacons.

The percentages below reflect **how frequently a beacon arrived if sent during that event**, as compared to the "startup" beacon that was sent as soon as the page's `<head>` was parsed.

This test was done on a **single site** so results from other sites **will differ**.

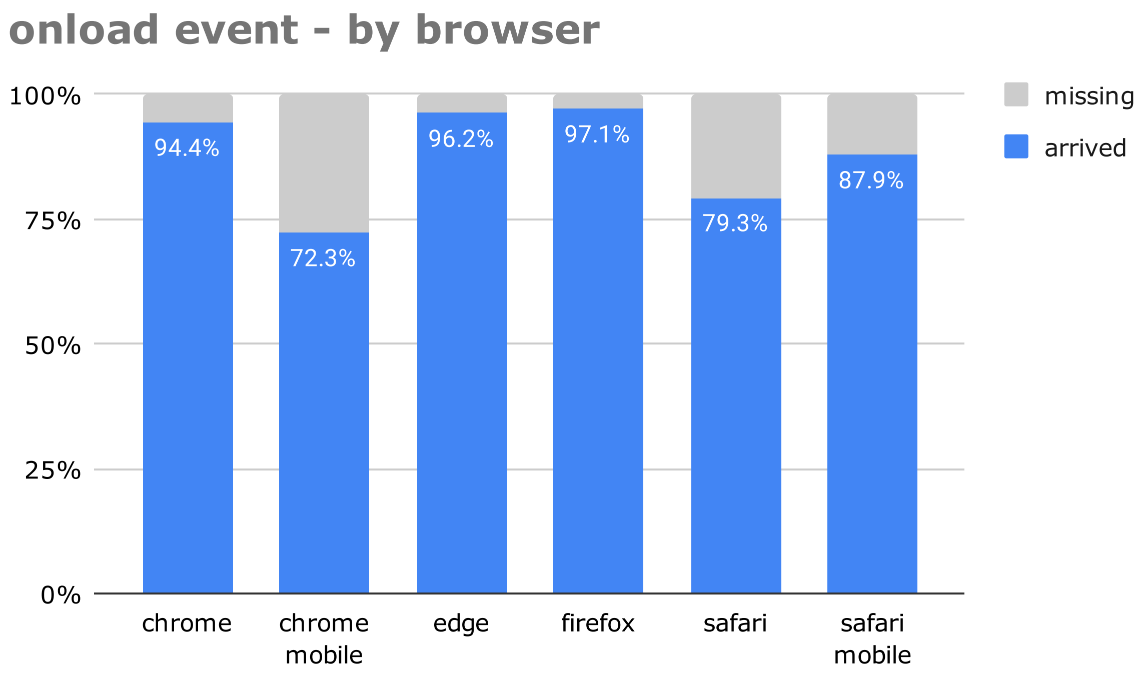
Page Load (`onload`) Event

Besides sending a beacon as soon as the page starts up, the most frequent opportunity to send data is the `window load` event (aka `onload`).



When sending data *just* at `onload`, beacons arrive only **86.4% of the time** (on this site).

This of course varies by browser:



A large percentage of those “missing” beacons are due to page *abandons*, i.e. when the visitor leaves before the `onload` event has fired.

This abandon rate will vary by site, but for this particular site, nearly 14% of visits would not be tracked if you only listened to `onload`.

Thus, if your data requires waiting until the `onload` event, you should also listen to page lifecycle “unload” events, to get the opportunity to send a beacon if the user is leaving the page. See avoiding abandons below.

Delayed Page Load (`onload`) Event

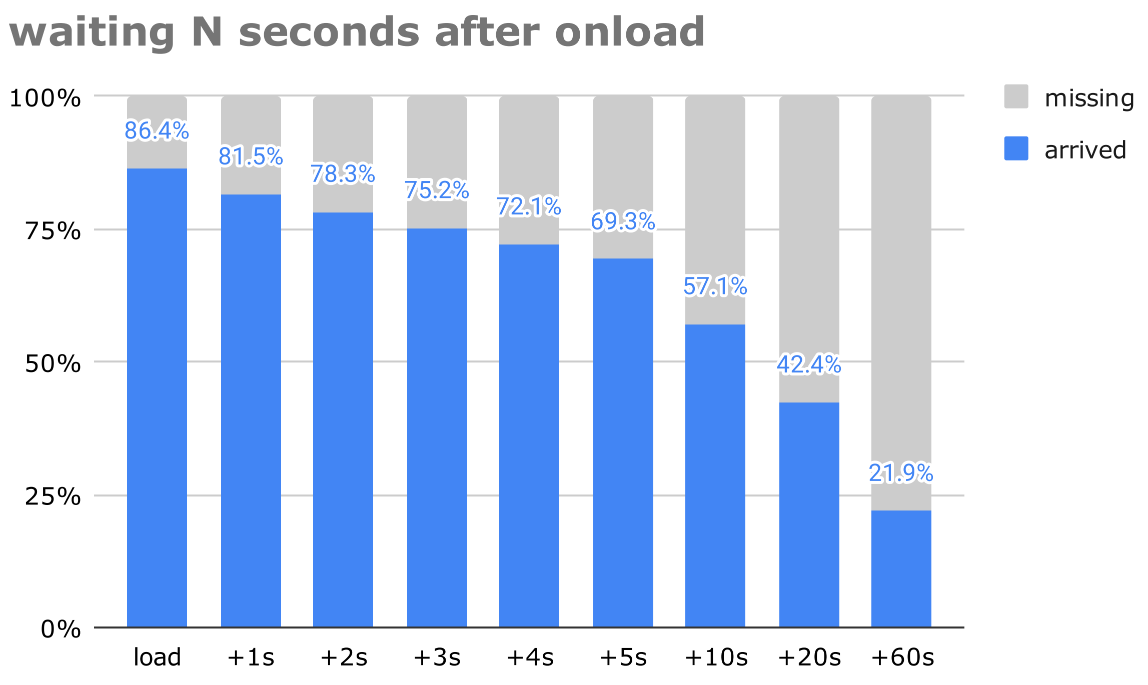
Sometimes, you may not want to send data immediately at the `onload` event. It could make sense to wait a little bit.

You could consider waiting a pre-defined amount of time, say 1 or 5 or 10 seconds after `onload` before sending the beacon.

Alternatively, if you have page components that are delay-loaded until the `onload` event, you may want to wait until they load to measure them.

Any amount of time you're waiting *beyond* the Page Load will **decrease beacon rates**, unless you're also listening to unload events (see below).

For example, artificially adding a delay after `onload` before sending the beacon resulted in a clear drop-off of reliability:



Again, these rates are if you *only* listen to the `onload` (and send a beacon N seconds after that) — you'd ideally pair this with avoiding abandons below to make sure you send a beacon if the visitor leaves first.

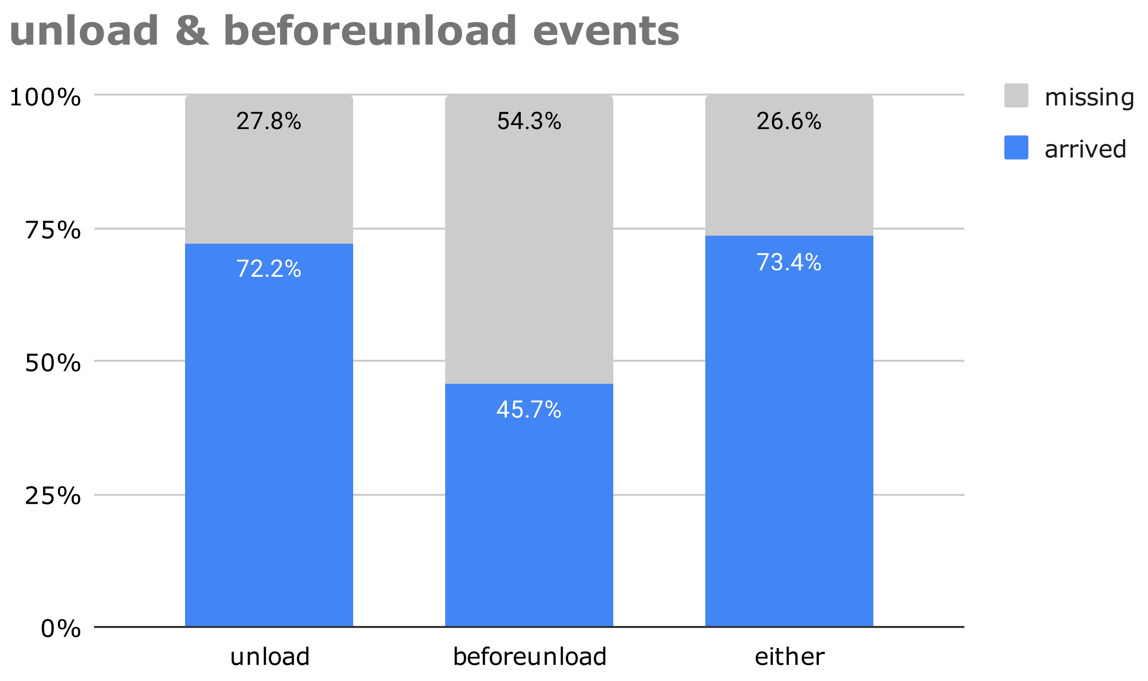
Unload Events

There are several events that are all related to the page "unloading", such as visibilitychange, pagehide, beforeunload, and unload. They are all used for specific purposes, and not all browsers support each event.

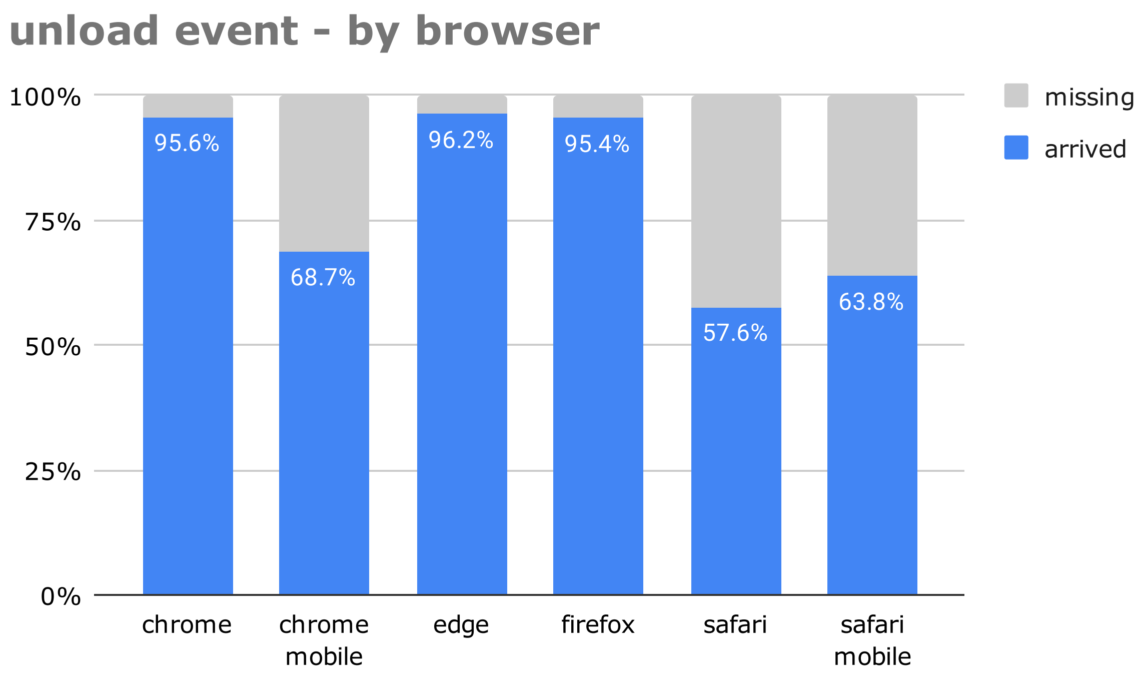
`unload` and `beforeunload` are two events that are fired as the page is being unloaded:

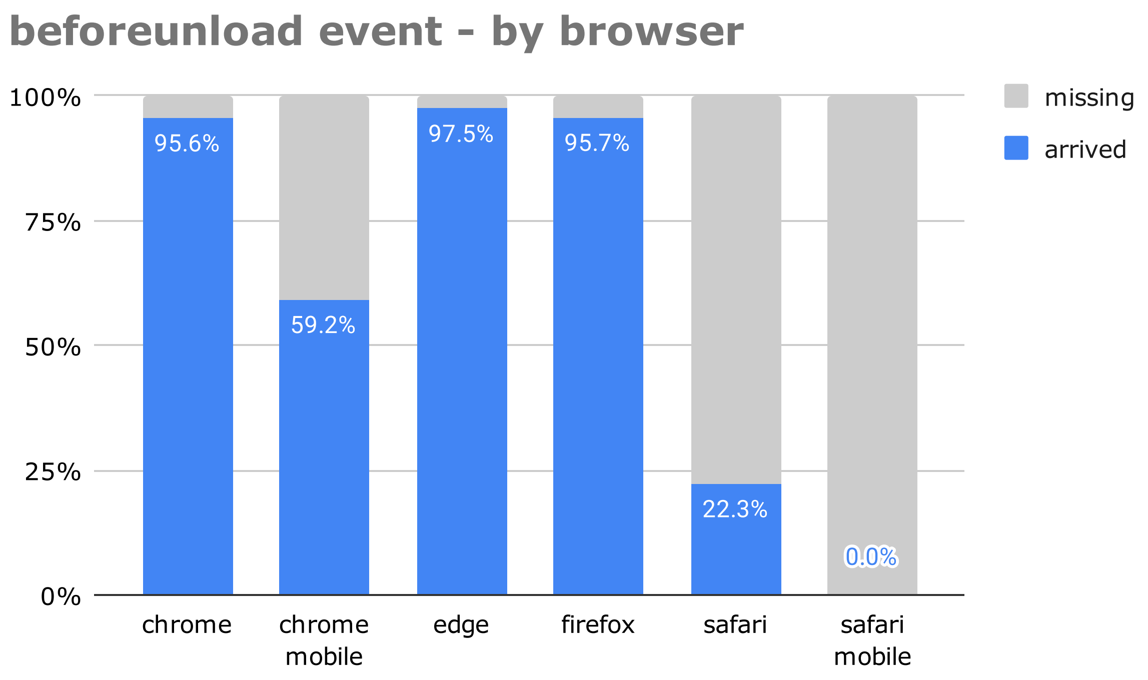
- `beforeunload` happens first, and gives JavaScript the opportunity to cancel the unload
- `unload` happens next, and there is no turning back

While the `unload` and `beforeunload` events have been with us since the beginning of the web, they're not the most reliable events to use for beaconing:



The `unload` event is significantly more reliable than the `beforeunload` event. This discrepancy is primarily due to browser differences:



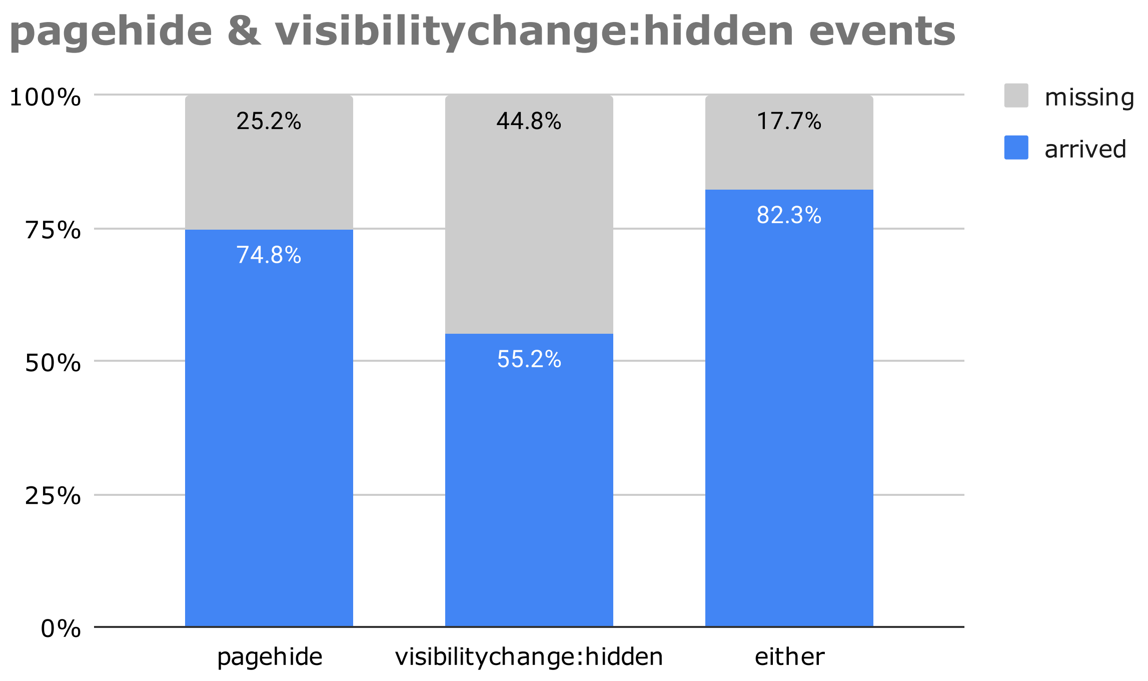


Notably, on Safari Mobile, `beforeunload` is not fired at all (while `unload` is).

`pagehide` and `visibilitychange` are more "modern" events:

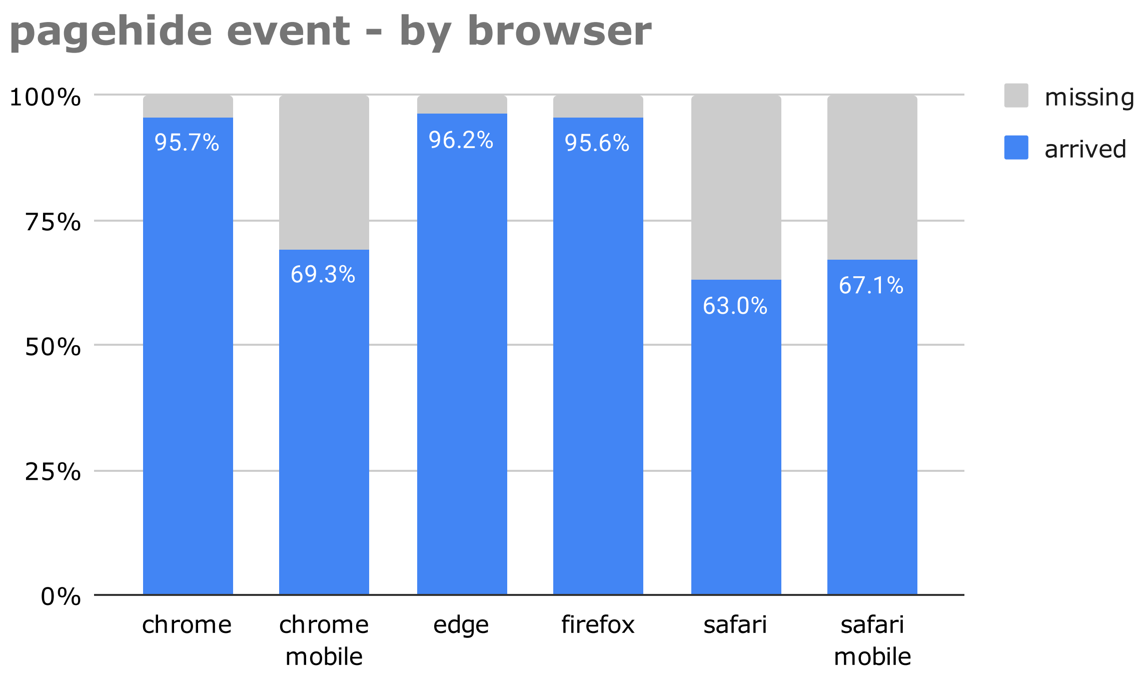
- `visibilitychange` can happen when a user switches to another tab (so the current tab is not *unloading* yet). This *may not* be the time you want to send a beacon, as a change to `hidden` doesn't preclude the page coming back to `visible` later — the user hasn't navigated away, just gone away (possibly) temporarily. But it's possibly the last opportunity you'll have to send data, so it's a good time to send a beacon if you can.
- `pagehide` was introduced as a more reliable "this page is going away" event than the original `unload` events, which have some caveats and scenarios where they aren't expected to fire.

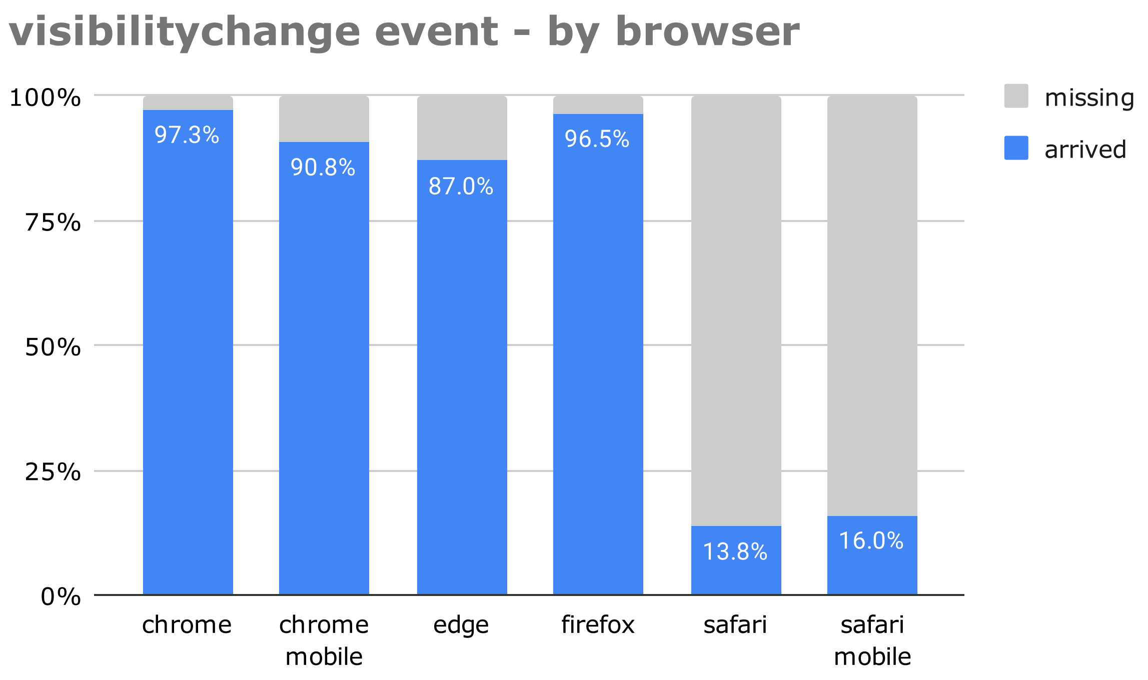
Here's how often beacons sent during those events arrived:



As seen above, we find `pagehide` (the modern version of `unload`) to be slightly more reliable than `unload` (74.8% vs. 72.2%). `visibilitychange` (hidden) alone doesn't send beacons as often, but if combined with `pagehide` events, we're up to 82.3% reliability which is superior to the combined 73.4% of `beforeunload|unload`.

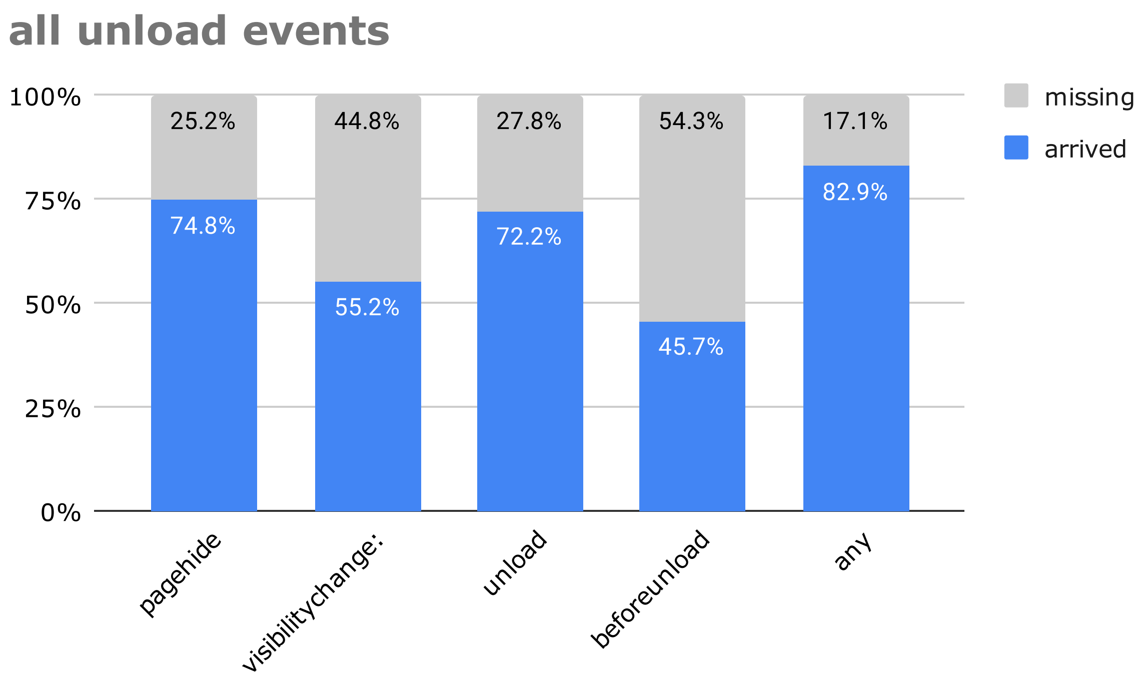
By browser:





Not coincidentally, listening for these two events `pagehide` and `visibilitychange` to save state or to send a beacon is the recommendation from Ilya Grigorik from back in 2015. This is still a great recommendation. However, if you're sending only a single beacon (and not just saving state), I recommend considering the trade-offs of attempting to beacon earlier in the process.

Below are all of the unload-style events in a single chart. If for some reason you want to listen to all of these events, you gain the most reliability (82.94%):



Listening to all events gives you 0.64% more reliability (82.94%) than just `pagehide`/`visibilitychange` (at 82.3%).

However, there is a major downside to registering for the `unload` handler: it breaks BFCache in Chrome , Safari and Firefox! BFCache is a browser performance optimization that's been available in Firefox and Safari for a while, and was recently added to Chrome 86+. The `beforeunload` handler also breaks BFCache in Firefox.

Depending on your site (or if you're a third-party analytics provider), you should consider the trade-off of more beacons vs. breaking BFCache when deciding which events to listen for.

Note: Not all browsers support `pagehide` or `visibilitychange`, so you'll want to detect support for those and if not, fallback to listening for `unload` and `beforeunload` as well.

Wrapping this all together, here's my recommendation for listening for unload-style events to get the most reliability:

```
// pseudo-code

// prefer pagehide to unload events
if ('onpagehide' in self) {
  addEventListener('pagehide', sendBeacon, { capture: true} );
} else {
  // only register beforeunload/unload in browsers that don't support
  // pagehide to avoid breaking bfcache
  addEventListener('unload', sendBeacon, { capture: true} );
  addEventListener('beforeunload', sendBeacon, { capture: true} );
}

// visibilitychange may be your last opportunity to beacon,
// though the user could come back later
addEventListener('visibilitychange', function() {
  if (document.visibilityState === 'hidden') {
    sendBeacon();
  }
}, { capture: true} );
```

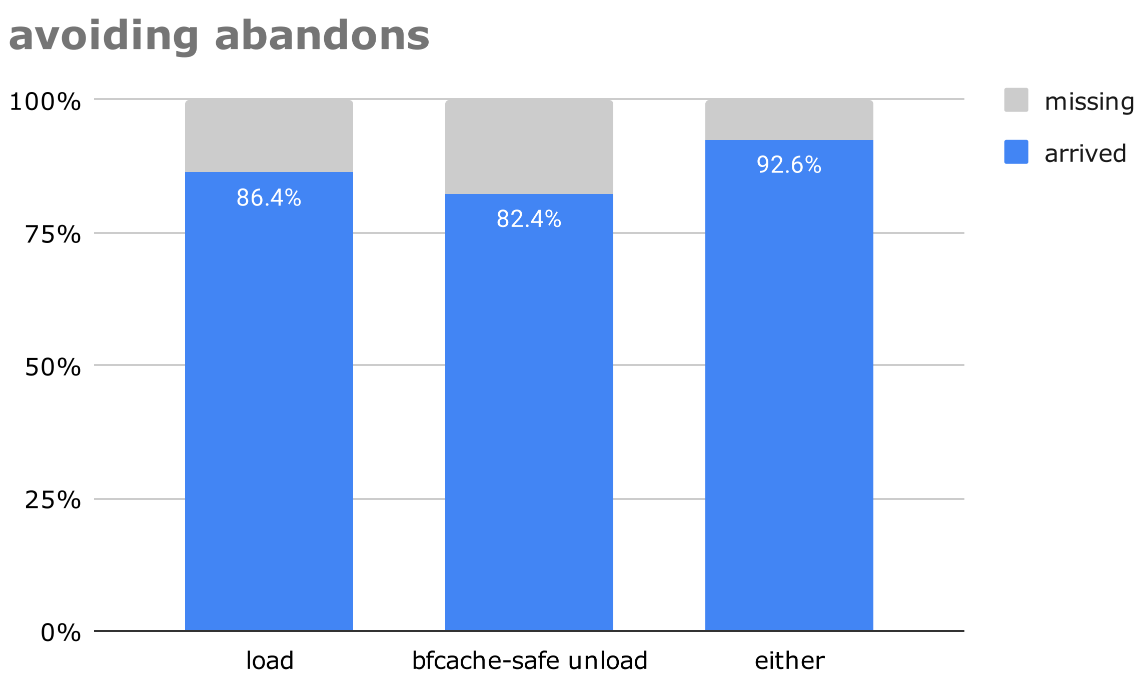
Avoiding Abandons

If your primary beaconing event is the Page Load (`onload`) event, but you want to also respond to users abandoning the page before the page reaches `onload`, you'll want to combine listening for both `onload` and Unload events.

When the page is abandoned prematurely, the page may not have all of the data you track for “full” navigations. However, there are often useful things you’ll still want to track, such as:

- That the Page Load happened at all
- Characteristics of the page, user, browser
- What “phase” of the Page Load they reached

Combining `onload` plus the two recommended Unload events `pagehide` and `visibilitychange` (hidden) gives you the best possible opportunity for tracking the Page Load:

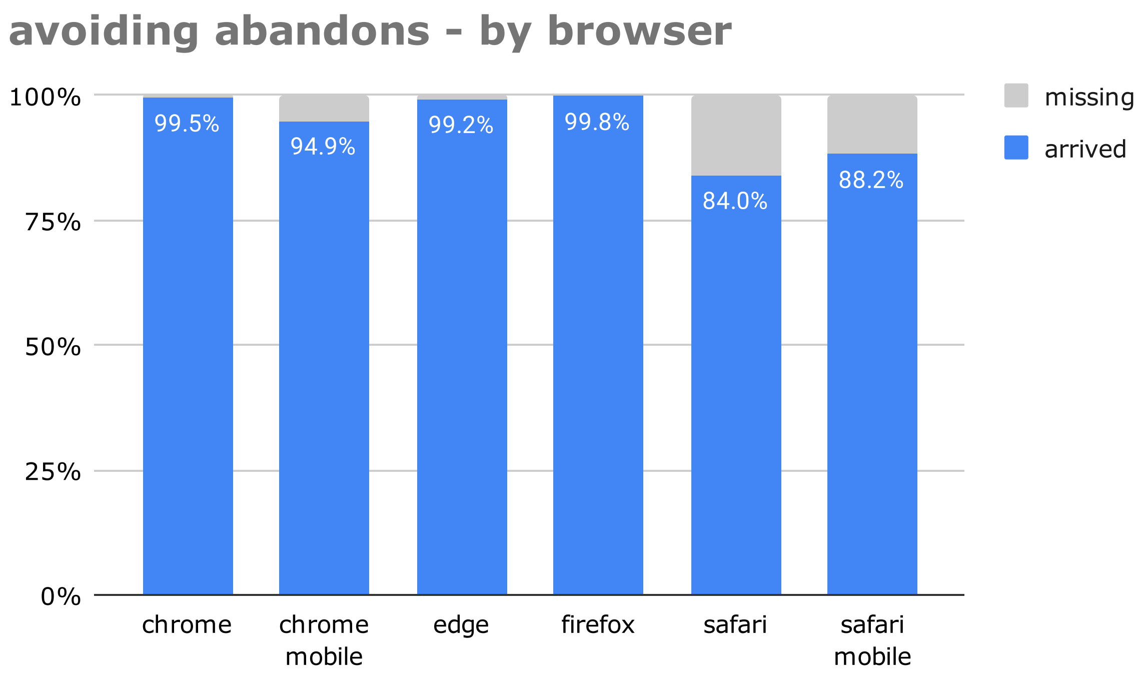


By listening to those three events, we see beacons arriving 92.6% of the time.

This rate:

- Decreases by just 0.6% to 92.0% if you don’t listen for `visibilitychange` (if you don’t want to beacon if the user might come back after a tab switch)
- Increases by just 0.2% to 92.8% if you listen for `beforeunload` (which would break BFCache in Firefox)
- Does **not increase** in any meaningful way if you also listened for `unload` (which breaks BFCache anyway).

By browser:



Notably Safari and Safari Mobile seem less reliably for measuring, likely due to not firing the `pagehide` and `visibilitychange` events as often.

So if your primary use case is just sending out **one beacon** by the `onload` (or Unload) event:

```
// pseudo-code

// prefer pagehide to unload event
if ('onpagehide' in self) {
  addEventListener('pagehide', sendBeacon, { capture: true} );
} else {
  // only register beforeunload/unload in browsers that don't support
  // pagehide to avoid breaking bfcache
  addEventListener('unload', sendBeacon, { capture: true} );
  addEventListener('beforeunload', sendBeacon, { capture: true} );
}

// visibilitychange may be your last opportunity to beacon,
// though the user could come back later
addEventListener('visibilitychange', function() {
  if (document.visibilityState === 'hidden') {
    sendBeacon();
  }
}, { capture: true} );

// send data at load!
addEventListener('load', sendBeacon, { capture: true} );

// track if we've sent this beacon or not
var sentBeacon = false;
function sendBeacon() {
  if (sentBeacon) {
    return;
  }

  // 1. call navigator.sendBeacon or XHR or Image
  // 2. cleanup after yourself, e.g. handlers

  sentBeacon = true;
}
```

One Beacon Trade-offs

Many analytics scripts prefer to send a single beacon. Taking boomerang as an example, we measure the performance of the user experience up to the Page Load (`onload`) event, and attempt to send our performance beacon immediately afterwards.

There are some continuous performance metrics, such as Cumulative Layout Shift (CLS) where it may be desirable to continue measuring the metric throughout the page's lifetime, right up to the unloading of the page. Doing so would track the "full page" CLS score, instead of just the CLS score snapshotted at the `onload` event.

There's an inherent trade-off when trying to decide to send a beacon immediately (at `onload`) instead of waiting until the unload event. Sending earlier is better for reliability, sending later is better for measuring "more" of the user experience.

Through this study we were able to quantify what this trade-off is (at least for the study's website):

- Sending a beacon at Page Load, with an abandonment strategy: 92.8% of beacons arrive
- Sending **only** a beacon at Unload: 82.3% of beacons arrive

So the "cost" of sending a single beacon at Unload instead of Page Load is about **10% of beacons don't arrive**. Depending on your priorities, that decrease in beacons may be worth measuring for "longer" before you send your data?

One important thing to remember when some beacons don't arrive is that their characteristics may not be evenly distributed. In other words, those 10% of beacons may be more frequently "good" experiences, or "bad" experiences, or a particular class of devices or browsers. Those missing beacons aren't a representative sample of the entire class of visitors, and could be hiding some real issues!

Bringing it back to Ilya's advice about saving app state via the unloading events: this is still suitable if you're saving app state or sending multiple beacons, but I'd suggest considering the reliability drop-off of not sending the beacon earlier, depending on the data you're measuring.

Advanced Techniques

If your goal is to capture as many user experiences as possible, there are a few more things you can try.

Persisting Beacon Data in Local Storage

If your goal is to send a single beacon, and you want to wait as long as possible to send it, you may want to only register for Unload events.

Since not beaconing earlier has a trade-off of being less reliable, you could consider temporarily storing your upcoming beacon data into `localStorage` until you send it.

If your Unload events fire properly and you're able to send a beacon, great! You can remove that data from `localStorage` too.

However, if your application starts up and finds orphan beacon data from a previous Page Load, you could send it on that page instead.

This works best if you're concerned about losing data for users navigating across your site — obviously if a user navigates away to another website, you may never get the opportunity to send data again (unless they come back later).

Service Workers

You could also consider using a ServiceWorkers as a "network buffer" for your beacon data.

If you're goal is to send a single beacon but want to wait until as late as possible, you can reduce some of the reliability trade-offs by "sending" the data to a ServiceWorker for the domain, and letting it transmit at its leisure.

You could have a communications channel with your ServiceWorker where you keep updating its beacon data throughout the page's lifetime, and rely on the ServiceWorker to send when it detects the user is no longer on the page

The reason this works is often a ServiceWorker will persist beyond the page's lifetime, even if the user navigates to another domain entirely. This won't work if the browser is closed (or crashes), but ServiceWorkers often live a little beyond the page unload.

Using a ServiceWorker would be best suited for first-party beacons (i.e. capturing data on your own site) — most third-party analytics tools would have a hard time convincing a domain to install a ServiceWorker just to improve their beacon reliability.

Misc

Cleanup

After you've successfully sent your data, it's a good opportunity to consider cleaning up after yourself if you don't anticipate any additional work.

For example, you could:

- Remove any event listeners, such as click handlers or unload events
- Discard any shared state (local variables)

You may not need to do this if you're sending a beacon as the result of an unload event firing, but if you're sending data earlier in the Page Load process, make sure you JavaScript won't continue doing work even though it'll never send a beacon again.

During Prerender or when Hidden?

You should consider whether it makes sense for you to send a beacon if the user hasn't seen the page yet.

The most likely scenario is when the page is loaded completely hidden. This can happen when a user opens a link into a new (background) tab, or loads a page and tabs/switches away before it loads.

Is this experience something you want to track? Does the experience matter if the user never saw the page? If you do want to send a beacon, do you send it at `onload` or wait until the page becomes `visible` first? These are all questions you should consider when capturing telemetry.

In Boomerang for example, we still measure those "Always Hidden" user experiences (where the user never sees the page before `onload`), and send a beacon right away. However, the beacon is also tagged with a special parameter, so the back-end (like mPulse) can "bucket" those user experiences so they can be excluded (or reviewed independently) from regular Page Loads.

There used to be some user agents that would also implement a "prerender" mode, but that was abandoned a few years ago. There's a new privacy-focused prerender proposal that may come back at some point that you should consider similar to the "hidden" case above.

The Future

Because of the limitations we mentioned in this article around the trade-offs for a "one beacon" approach versus its reliability, there have been recent discussions around using something like the Reporting API as a better "beacon data queuing mechanism" that would reliably send your beacon data when the user leaves the page.

You can see a presentation from Yoav Weiss from this year's 2020 W3C WebPerf TPAC event.

This could enable better capturing of continuous metrics (like CLS) via a single beacon sent *just* at the end of the Page Load in a reliable way.

Hoping the discussion continues!

TL;DR Summary

There are many reason *why* and *when* you may want to send beacons, but here are some high level tips:

- Use `navigator.sendBeacon()` when possible, but listen to its return codes and fallback to `XMLHttpRequest` or `Image` beacons when needed
- Send your beacon(s) as early as possible to ensure as many can reach your endpoints
- If you're waiting for a specific event to send your beacon, like Page Load, make sure you also have an abandonment strategy
- There are several browser events that happen near the unloading of a page — listen to `pagehide` and `visibilitychange` (hidden) (and not `unload` or `beforeunload` which can break BFCache)
- Be aware of your content and look for ways of minimizing payload size via compression or other means if it makes sense

Finally, we started this research by looking into our own beaconing strategy in Boomerang. We've found a few key changes we should make:

- We currently listen for the `unload` and `beforeunload` events to try to make sure we capture all abandons/unloads. This is not only unnecessary (it does not meaningfully increase reliability rate), it also breaks BFCache in nearly all modern browsers
- We do not currently listen for `visibilitychange` (hidden) to send our beacon, and we should consider it as it would increase our reliability (by 0.6% points)
- Boomerang generally sends its Page Load beacon right at `onload` if possible, as we were concerned with losing measurements if we waited later. This study found we'd miss around 10% of all Page Loads if we only sent our beacon during Unload instead. This may be a tradeoff some RUM customers want, so we can add that as an option.

- 2011
- 2010
- 2009