SERVICE WORKERS

& The Cache API

By definition a **Progressive Web App** must work offline.

This is achieved by caching the files you need for your application and using the cached versions (when appropriate) so that

1) your application will be faster

2) your application can launch when it is offline.

JavaScript provides an API for you to manage a cache specifically for this purpose.

This is separate from the regular browser cache (and is not effected by techniques for managing that cache)

To simplify the use of that cache Service Workers were introduced.

The service worker is a JavaScript file that can intercept network requests and decide what to return in reaction to them.

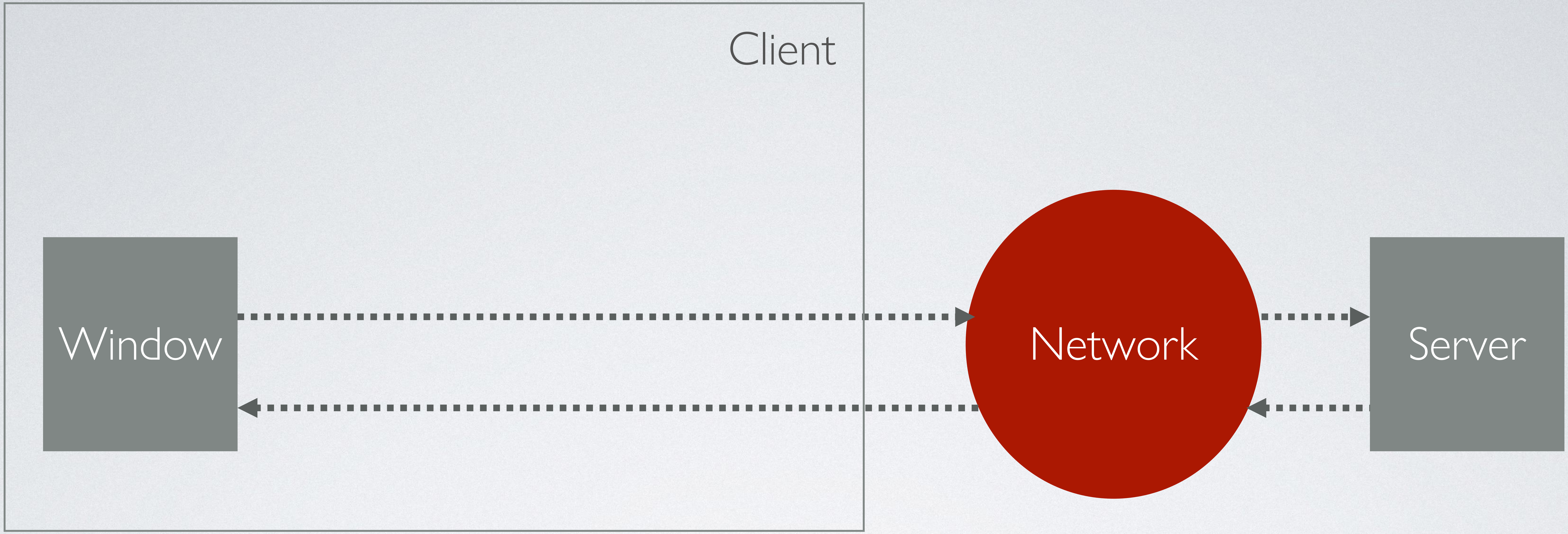
It can decide whether to return:

1) the originally **requested file** over the network (assuming it can be reached)

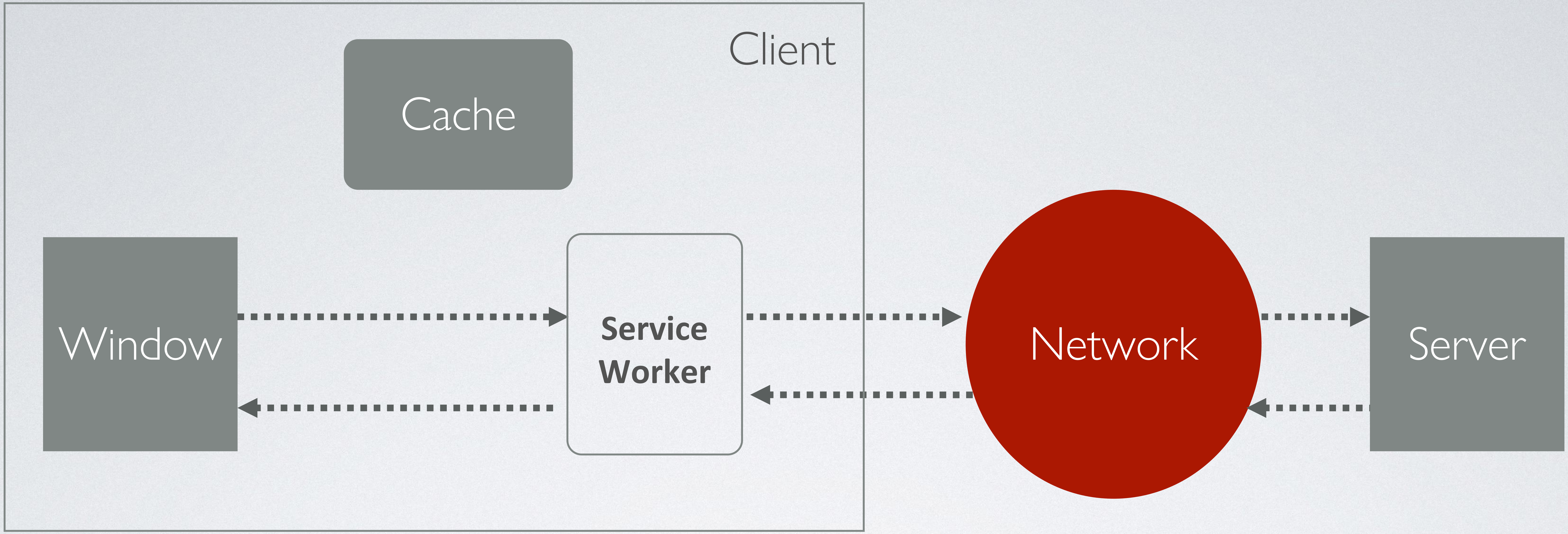
2) the version of that file in the **cache** (assuming it is in the cache)

3) an entirely **different file** (usually contained in the cache)

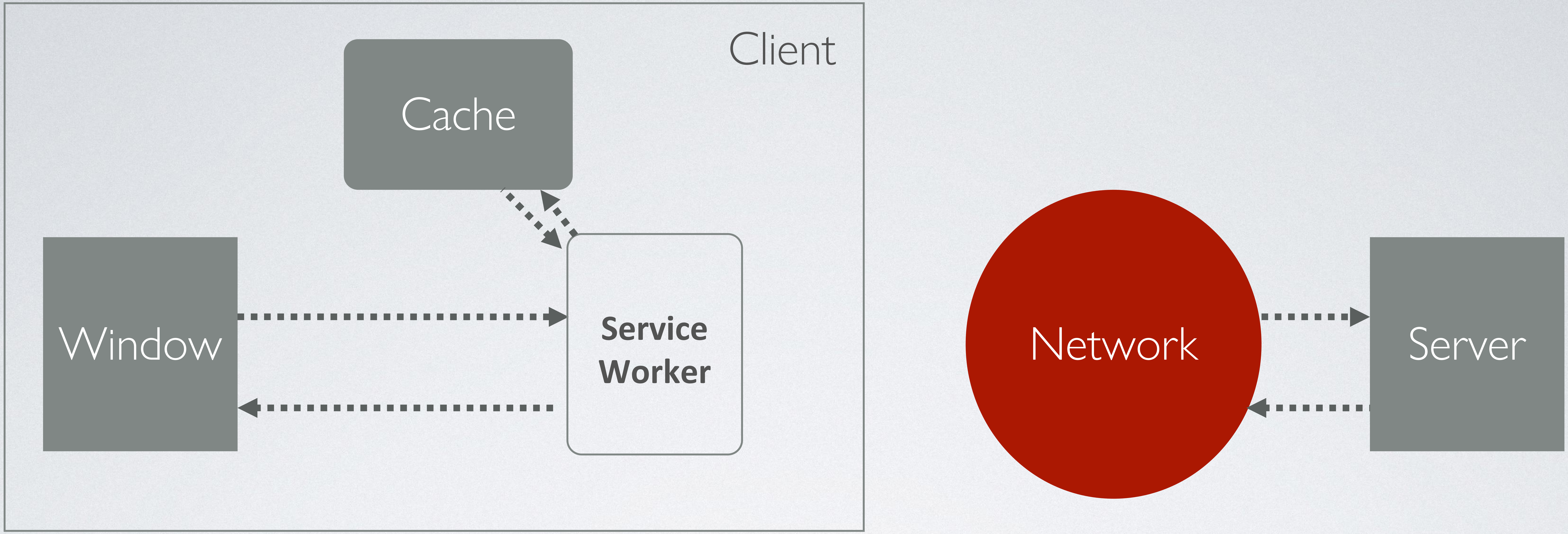
etc.



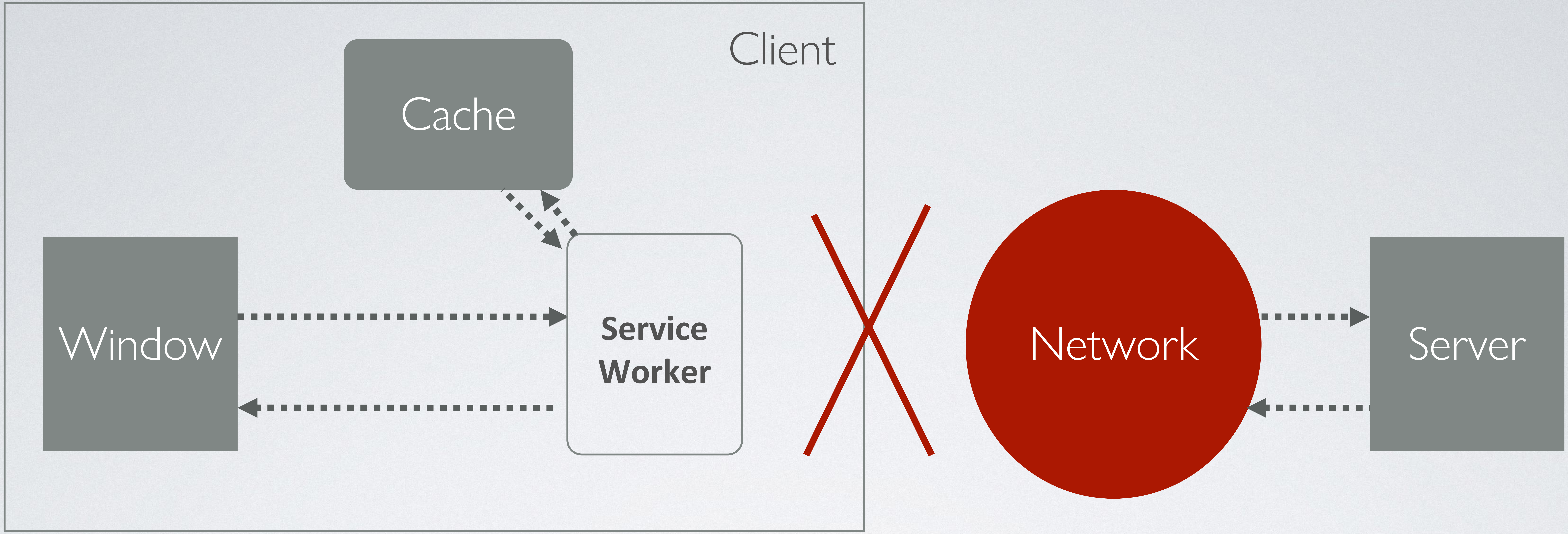
Typically, a web page running in a client will look for resources by accessing the network, and the server that hosts those resources.



A service worker is a script that sits between your web page, running in a window/tab, and the network. Every request your web page makes for a network resource must go through the service worker.

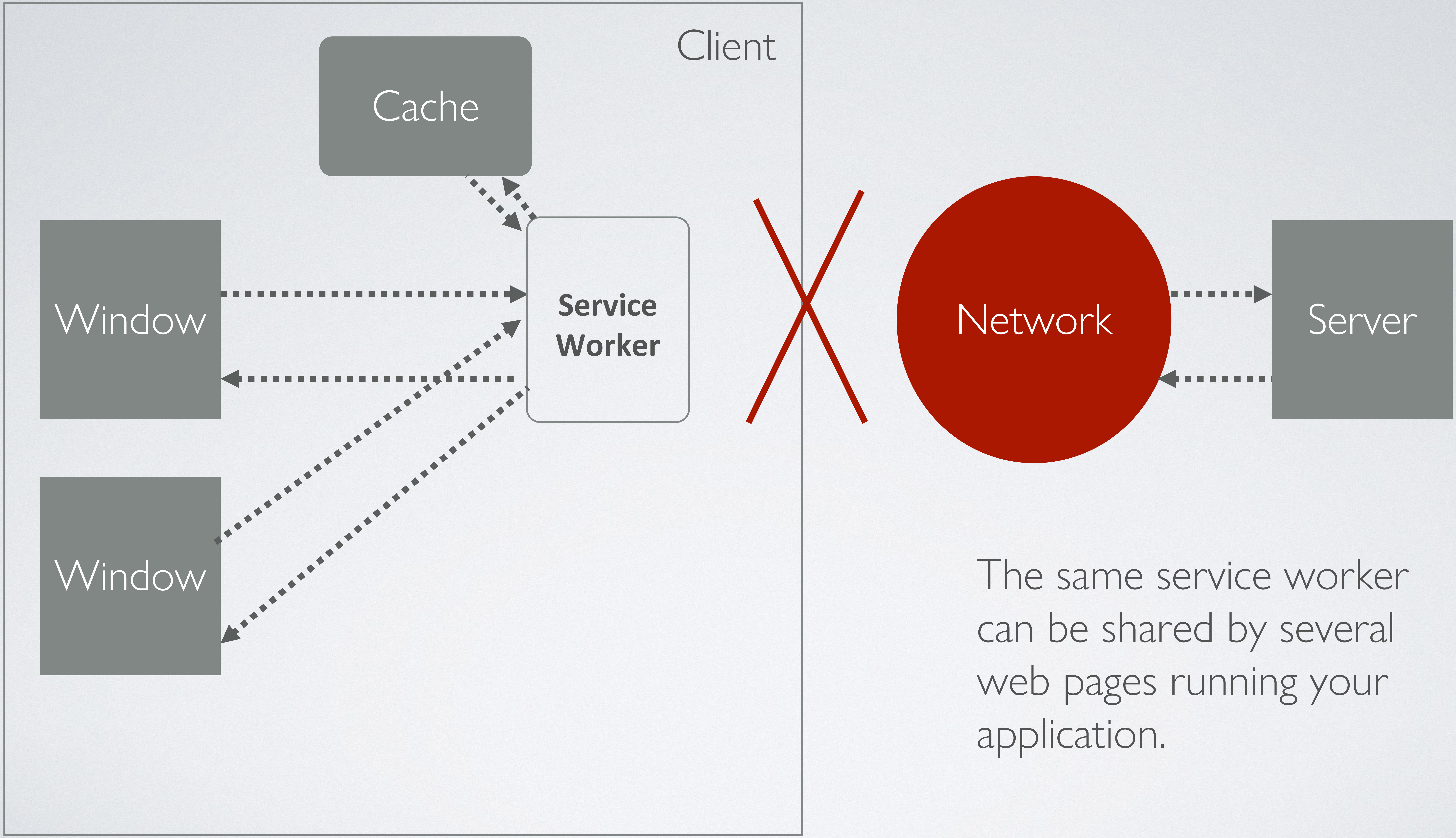


The Service Worker could choose to retrieve the files from a local cache instead (if it has local copies). These files would be available much sooner than files retrieved over the internet.

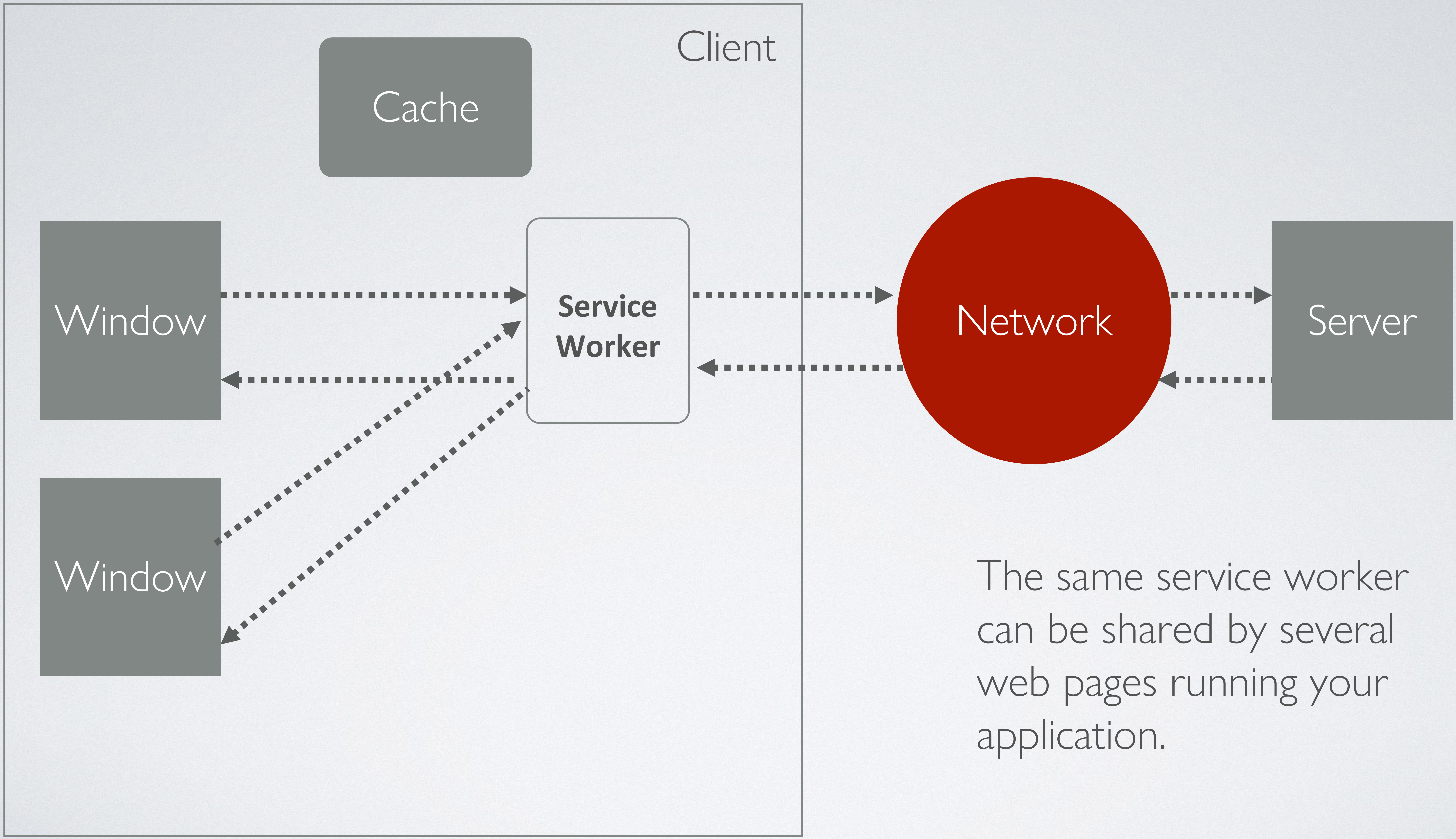


Similarly, if you can't reach the network the service worker can retrieve the resources from the cache instead. I.e. your app can work if you are offline.

The user need not notice any difference.



The same service worker can be shared by several web pages running your application.



The same service worker can be shared by several web pages running your application.

The service worker manages the cache. It usually decides what to put in the cache (and when). It decides whether to retrieve files from the cache or the network. It can also remove files from the cache.

(Note that the cache can be manipulated from within the window running your apps as well, i.e. in your regular JavaScript code)

Service Workers, like Web Workers, are separate JavaScript (.js) files containing code that provide functionality used by your regular JavaScript files.

Your regular JavaScript must register a Service Worker in order to make use of it.

Registering a Service Worker

In your webpage you need to specify the JavaScript file you want to use as your service worker.

You do this with the **serviceWorker** property of the **navigator** object.

```
navigator.serviceWorker.register('/serviceW.js');
```

It is advisable to check if service workers are supported by your device*.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/serviceW.js');
  });
}
```

or

```
window.addEventListener('load', function() {
if ('serviceWorker' in navigator) {
   navigator.serviceWorker.register('/serviceW.js');
  });
}
```

*Service Workers in iOs were only supported in March 2018

Service Worker Life Cycle

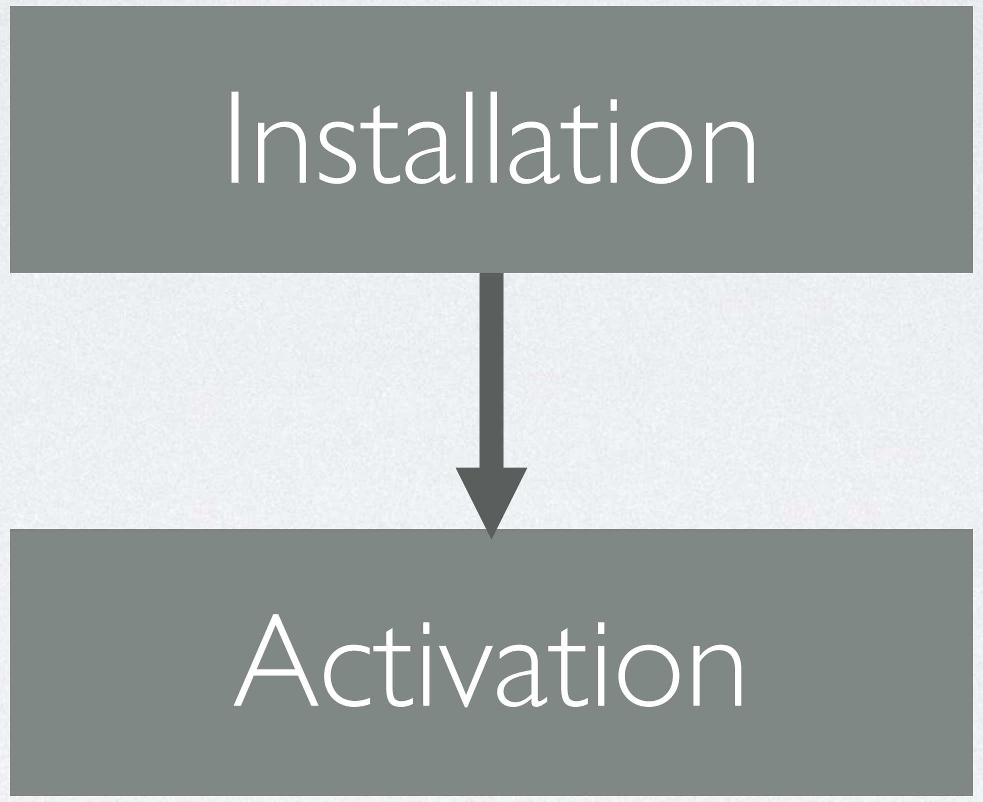
A service worker has a distinctive life cycle made up 3 phases

Installation

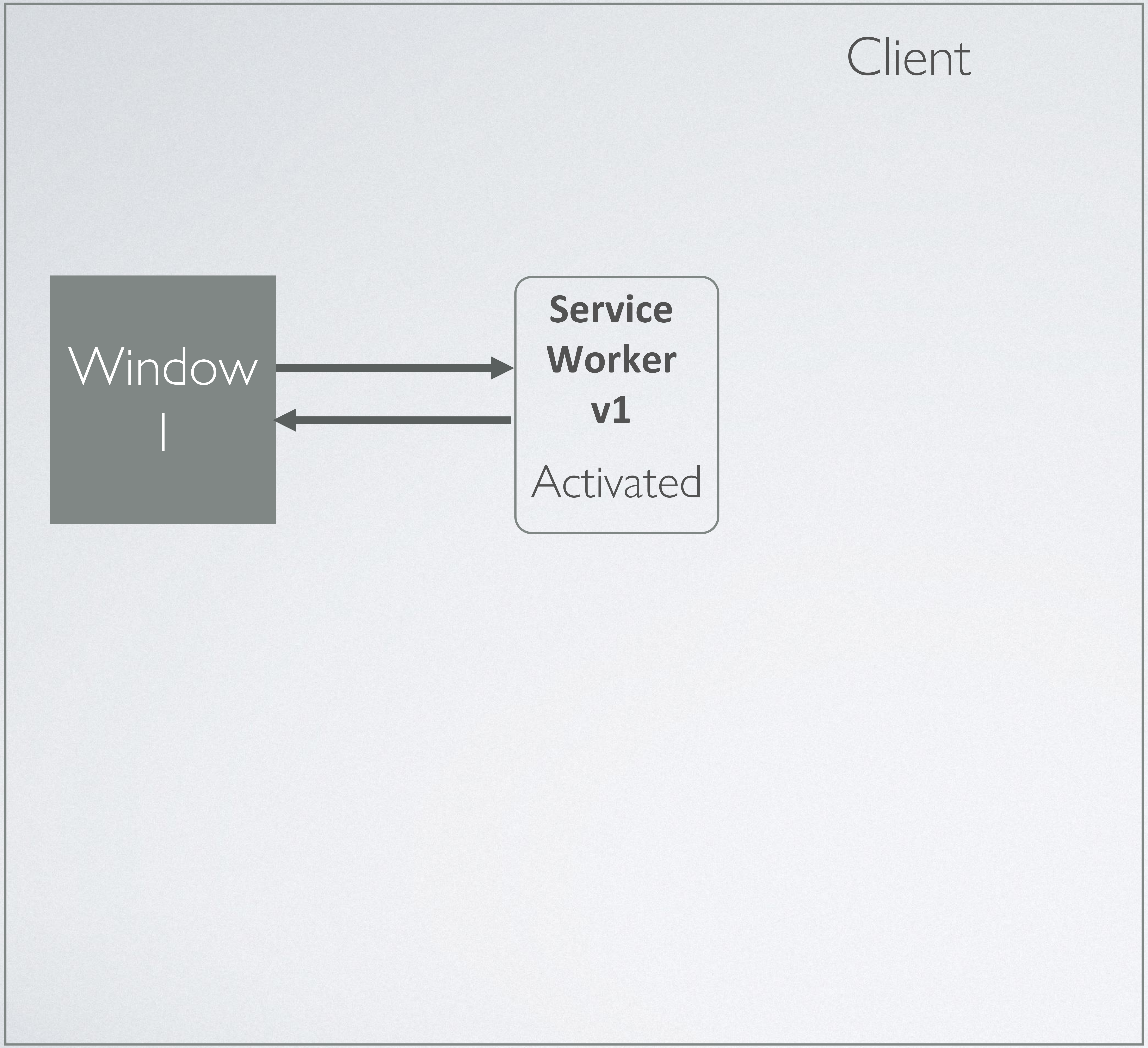
When your service worker is first registered it enters the **installation** phase.

This phase **occurs only once** for the service worker and can be used to prepare resources such as the cache.

The service worker then has to wait until it is activated.

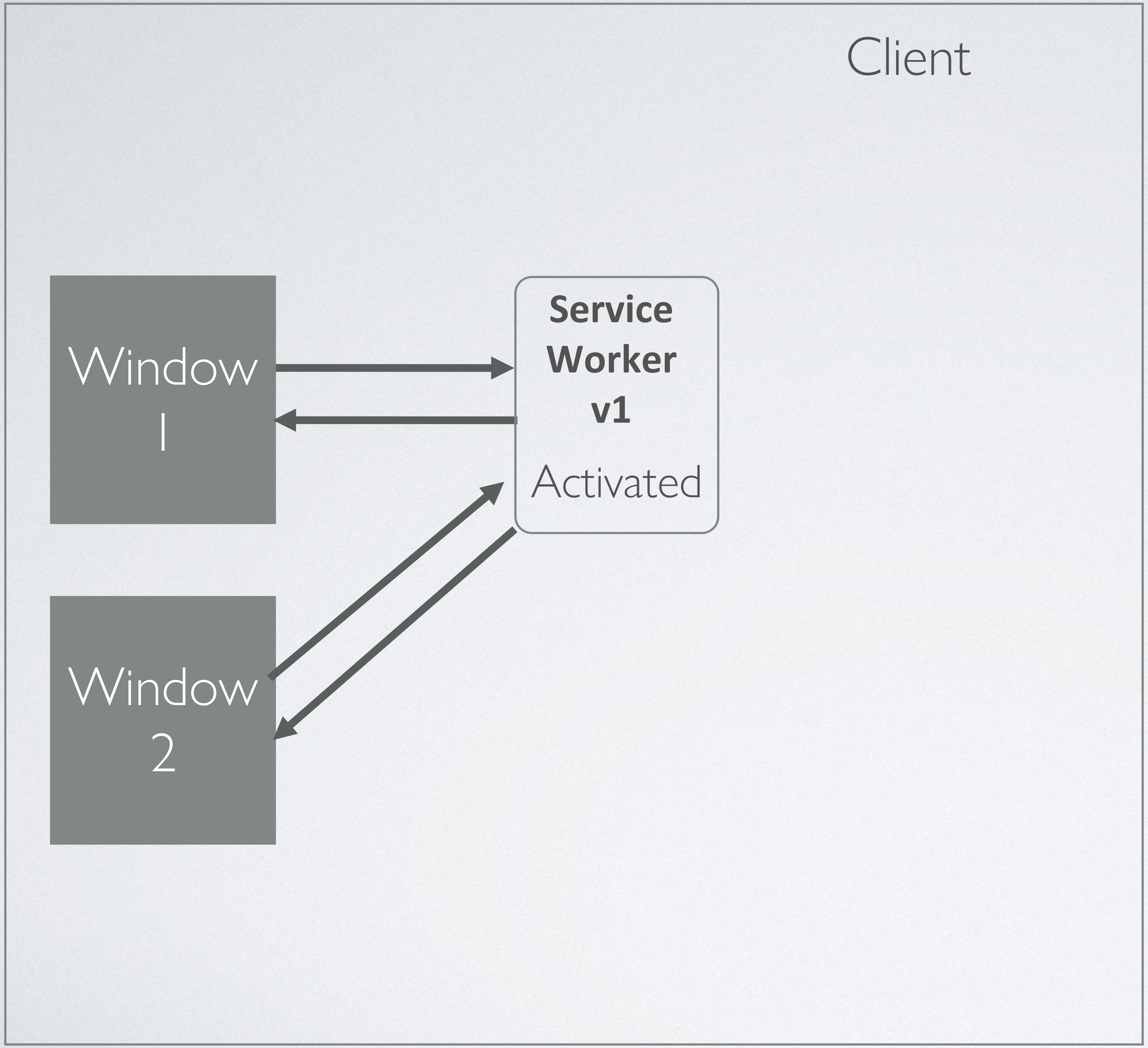


Once the service worker is active it can perform tasks that it could not beforehand.



To understand why there are two phases for setting up the service worker you need to know **when the service worker is activated**.

For the first service worker, the installation and activation phases happen in quick succession, since there is nothing to delay activation.

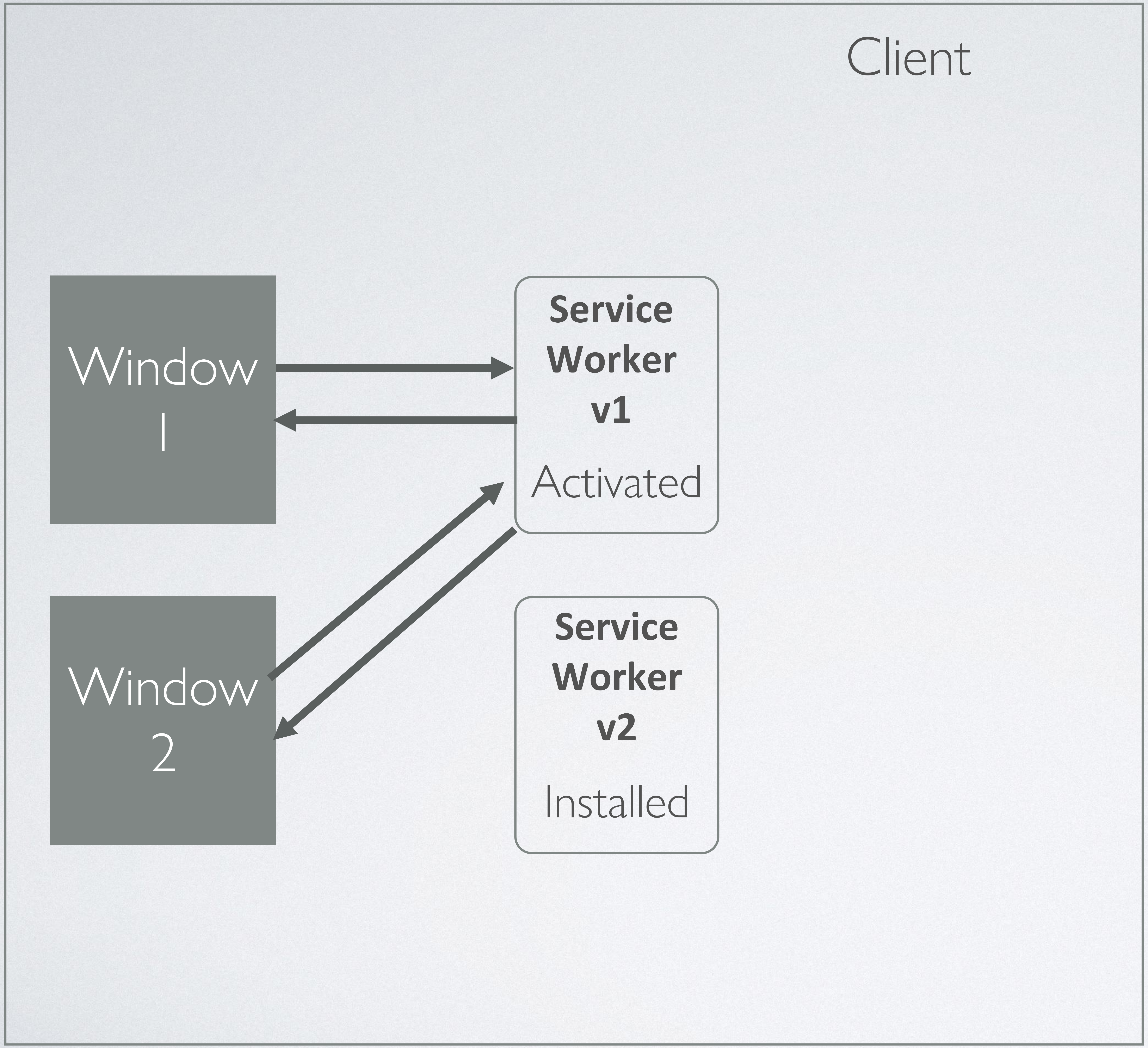


Now assume two pages are sharing the same service worker.

Frequently you might have to change some part of the service worker: either for a major update to the way your PWA functions, or for general maintenance of your code.

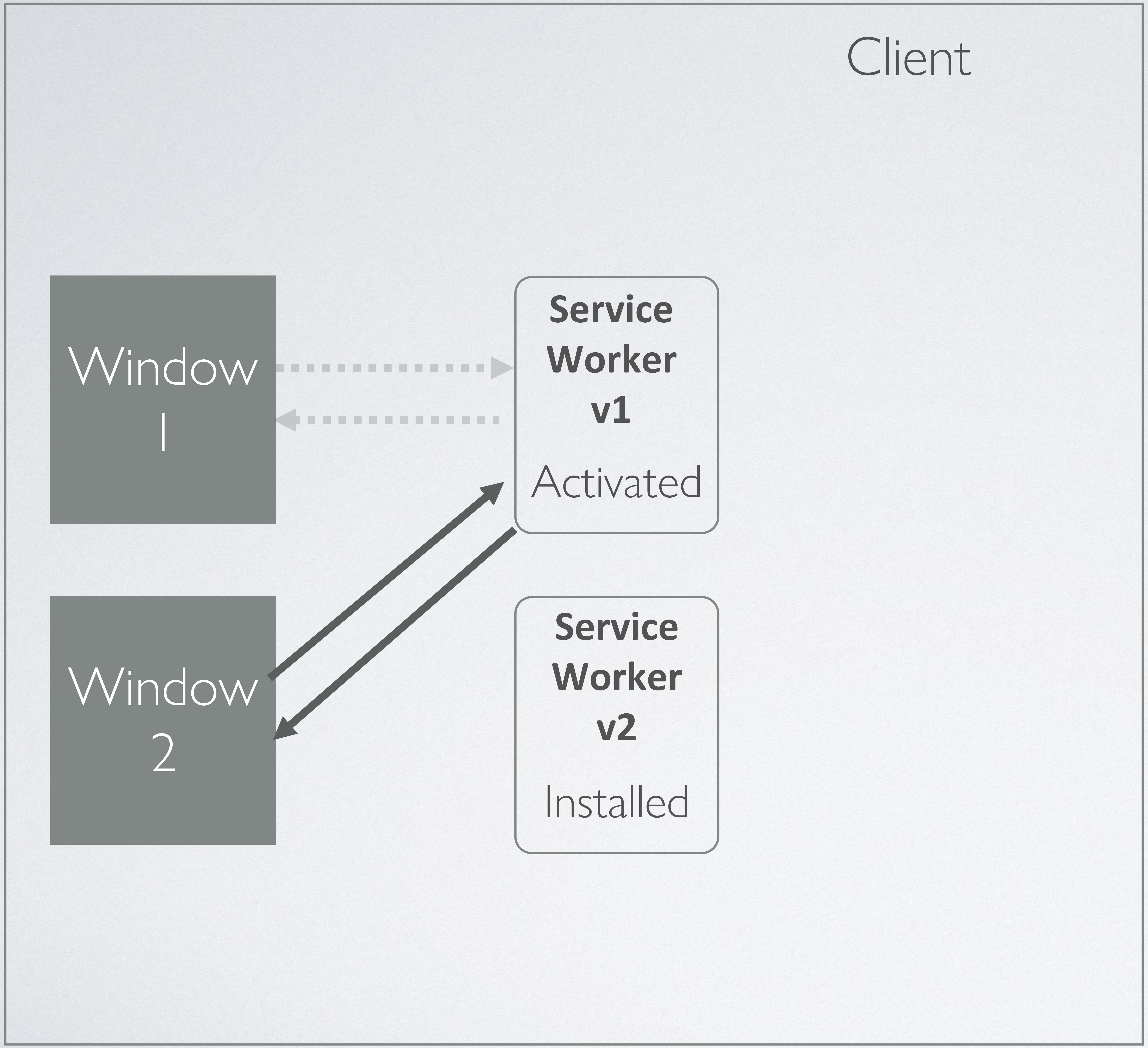
To make these changes you need only change the file on the server. Your original service worker will download the new version (as it does all files).

Now you have two service workers.



I.e. if you do update the service worker on the server the client downloads the new version but is still using the original version.

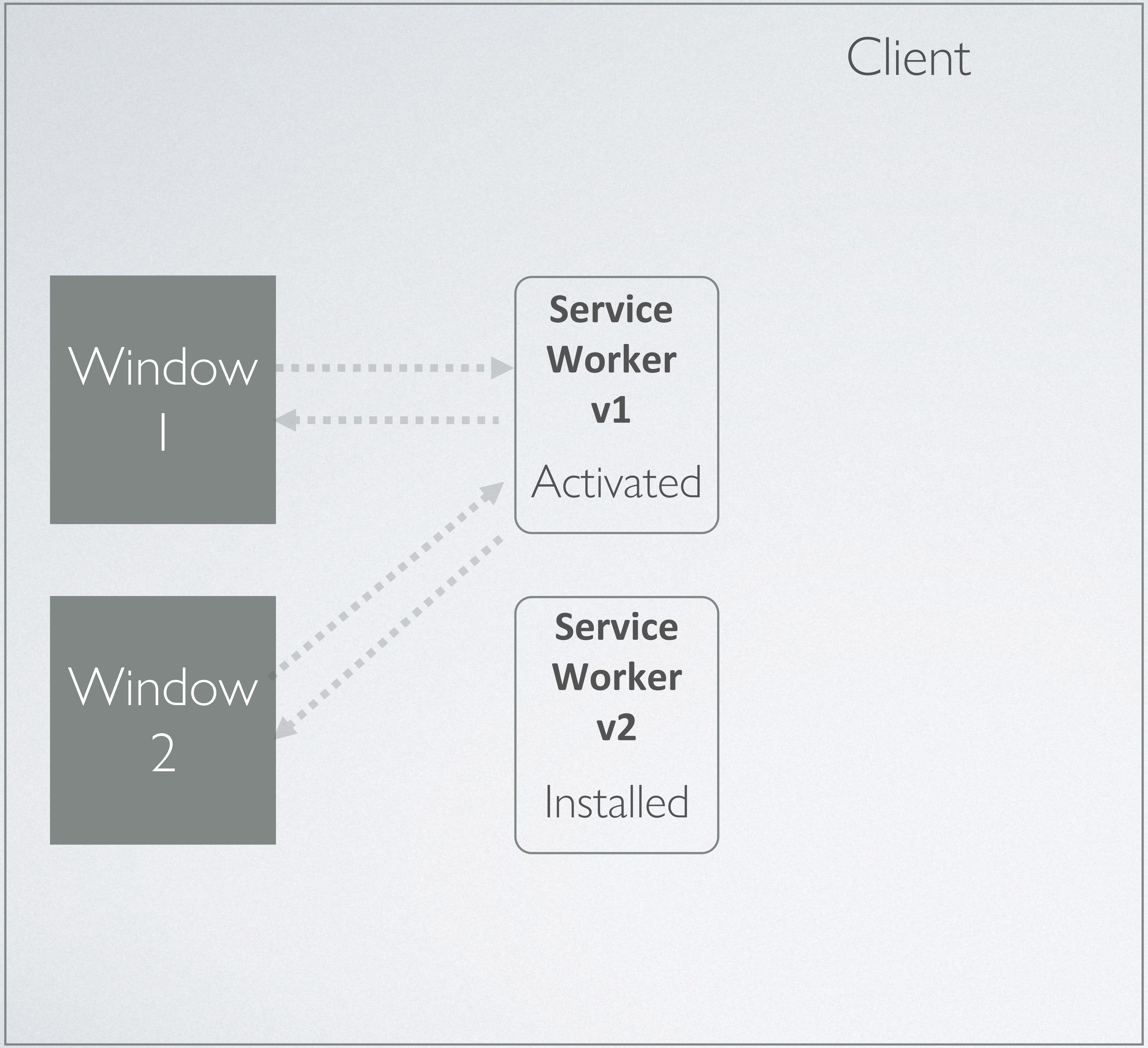
Even though is it not being used, the updated version **will still run its installation code**.



Assume Window 1 is not using its original service Worker (e.g. it is not currently requesting files).

However, it still can't start using the updated version.

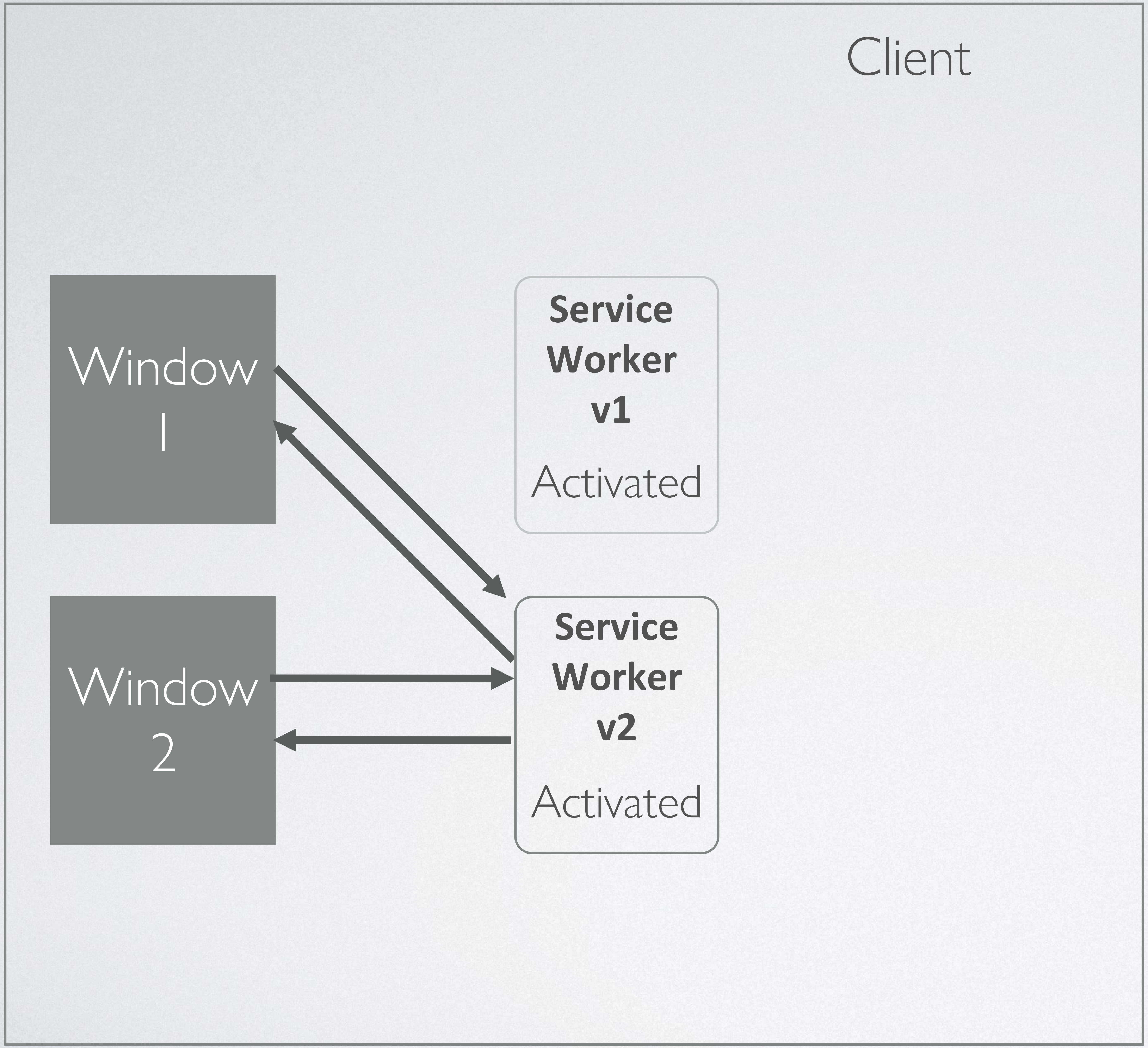
This is because **all pages must stop using** the original version before the new version can be used.



This is to ensure your application is executing consistently across all windows.

I.e. **they must all be using the same service worker at the same time**.

Eventually all windows/tabs will be stop using the original version.



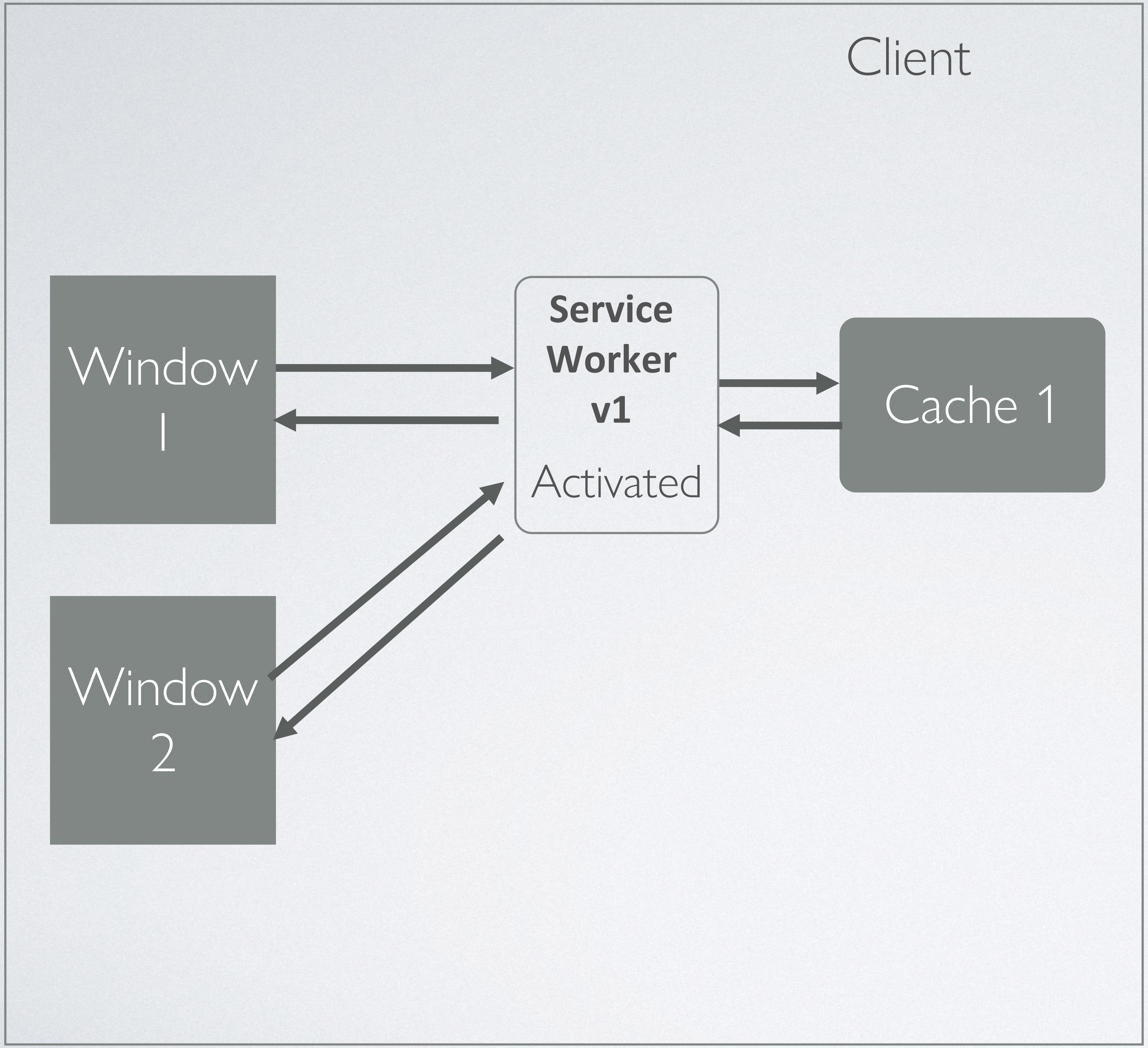
Now all the windows/tabs can be switched to using the new updated Service Worker.

So only now does that Service Worker enter the **activation** phase.

The reason the setup of the service worker is split into 2 phases is because of this delay in using the service worker

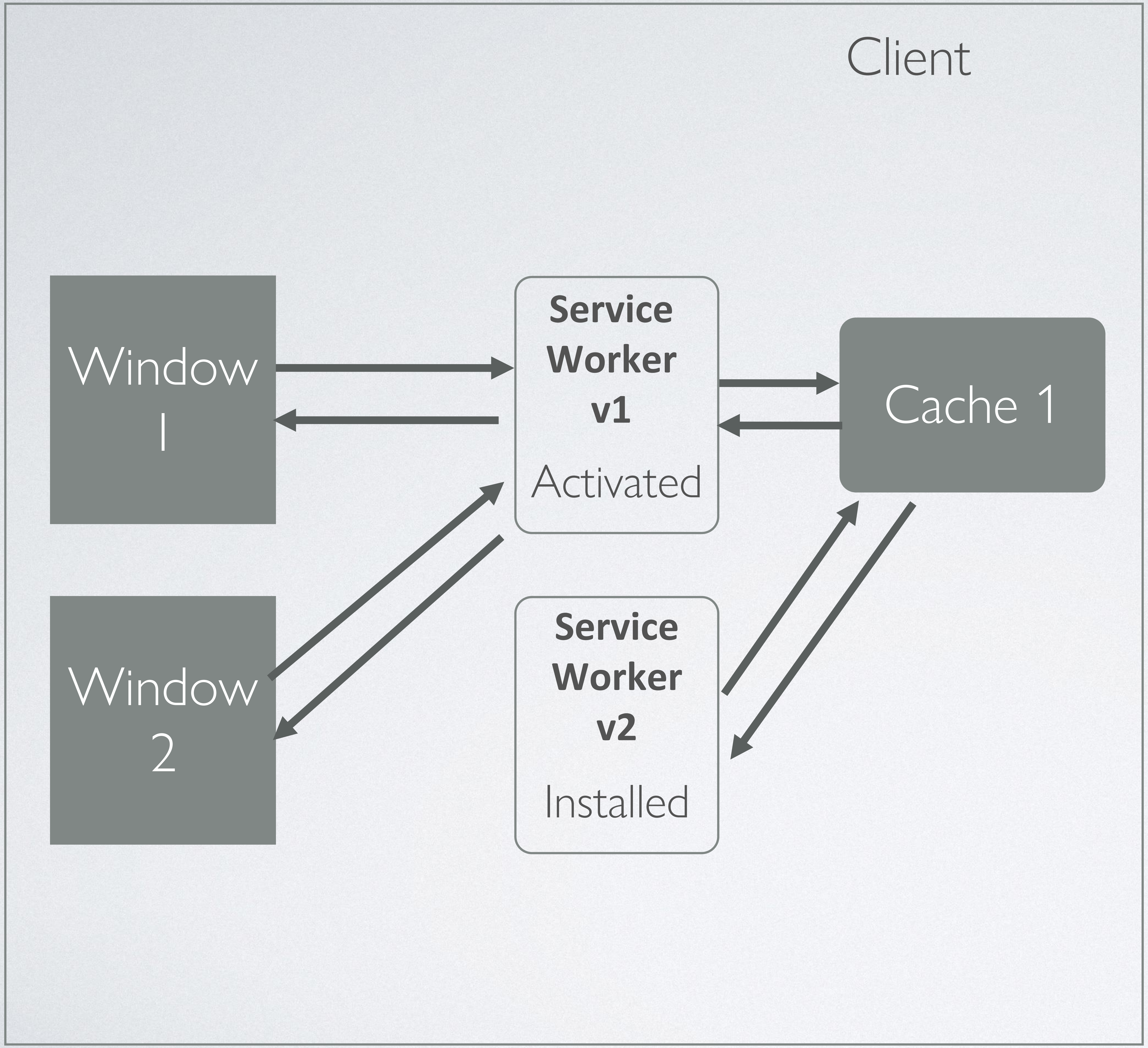
I.e. the new, updated, version of the service worker can't be expected to update the cache if the cache is being currently used by the original service worker (it is obviously a hazard if 2 independent service workers are trying to manage the same cache).

The activation phase only occurs ***after*** the previous service worker is no longer in use. I.e. when we know it is safe to access the cache (knowing no other service worker is).



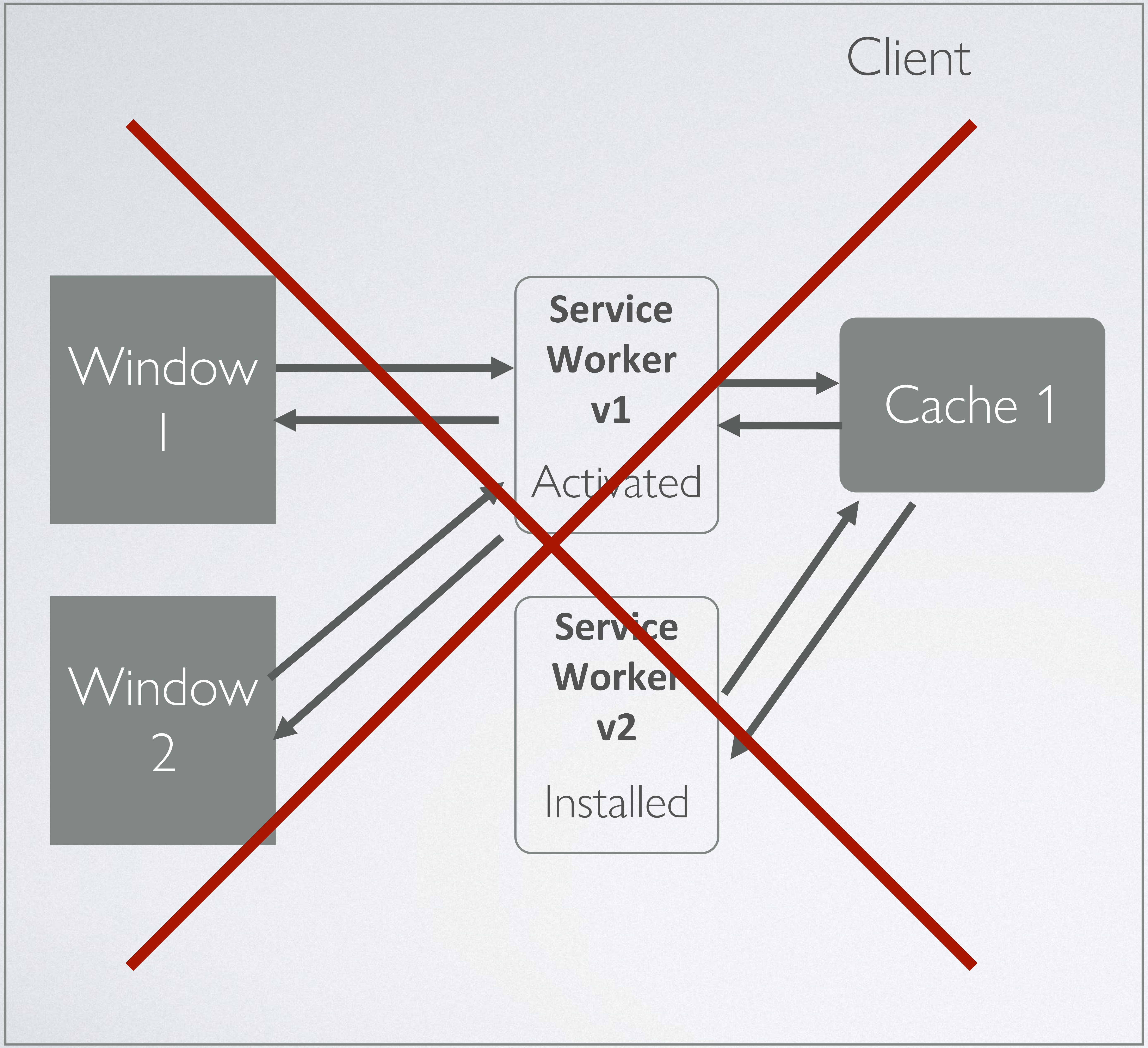
Consider this scenario.

The original service worker is using a cache.

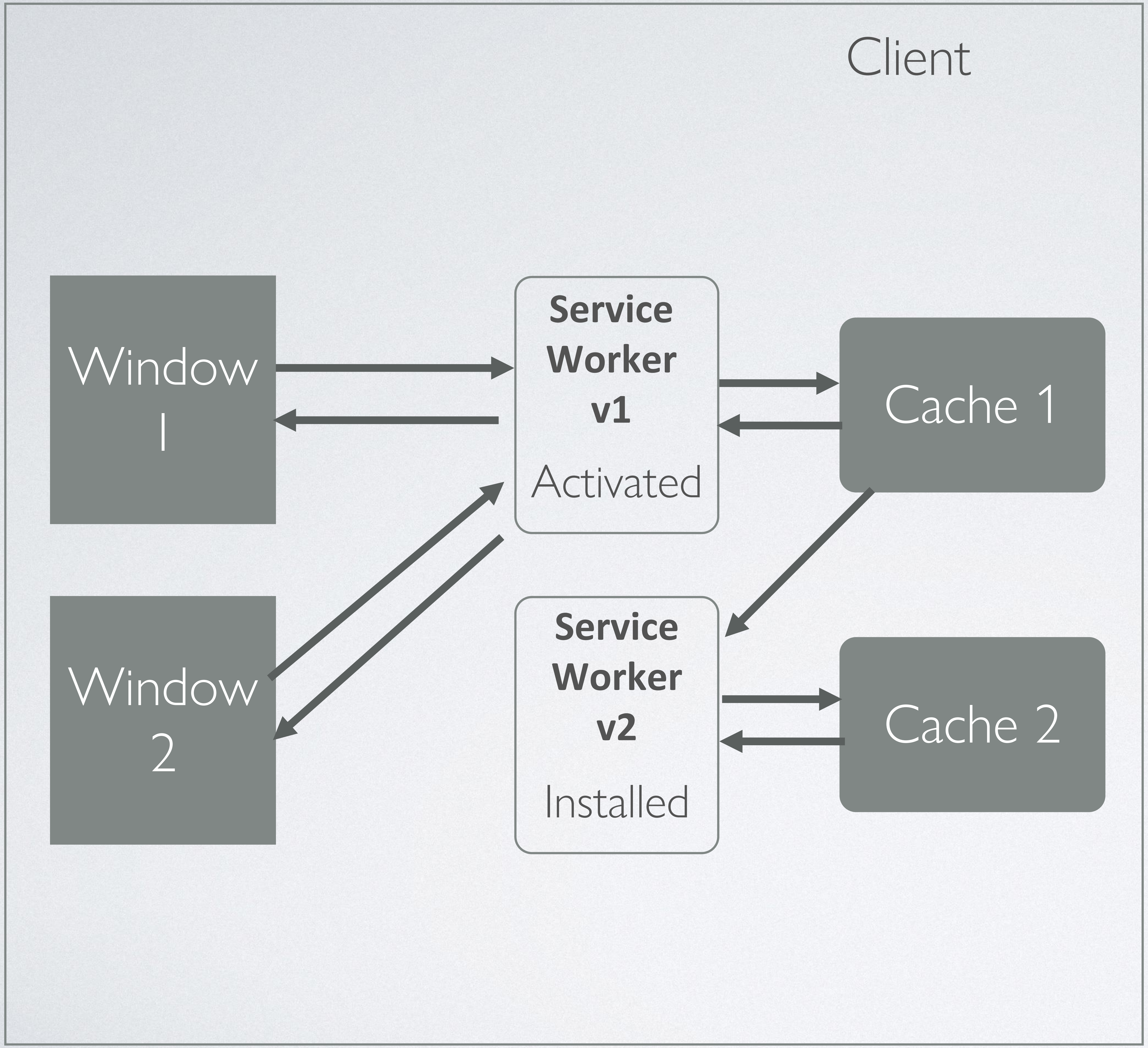


The updated service worker downloads and starts installing itself by preparing the cache (adding and deleting items).

But this is the cache being used by the original service worker, and therefore could cause problems.



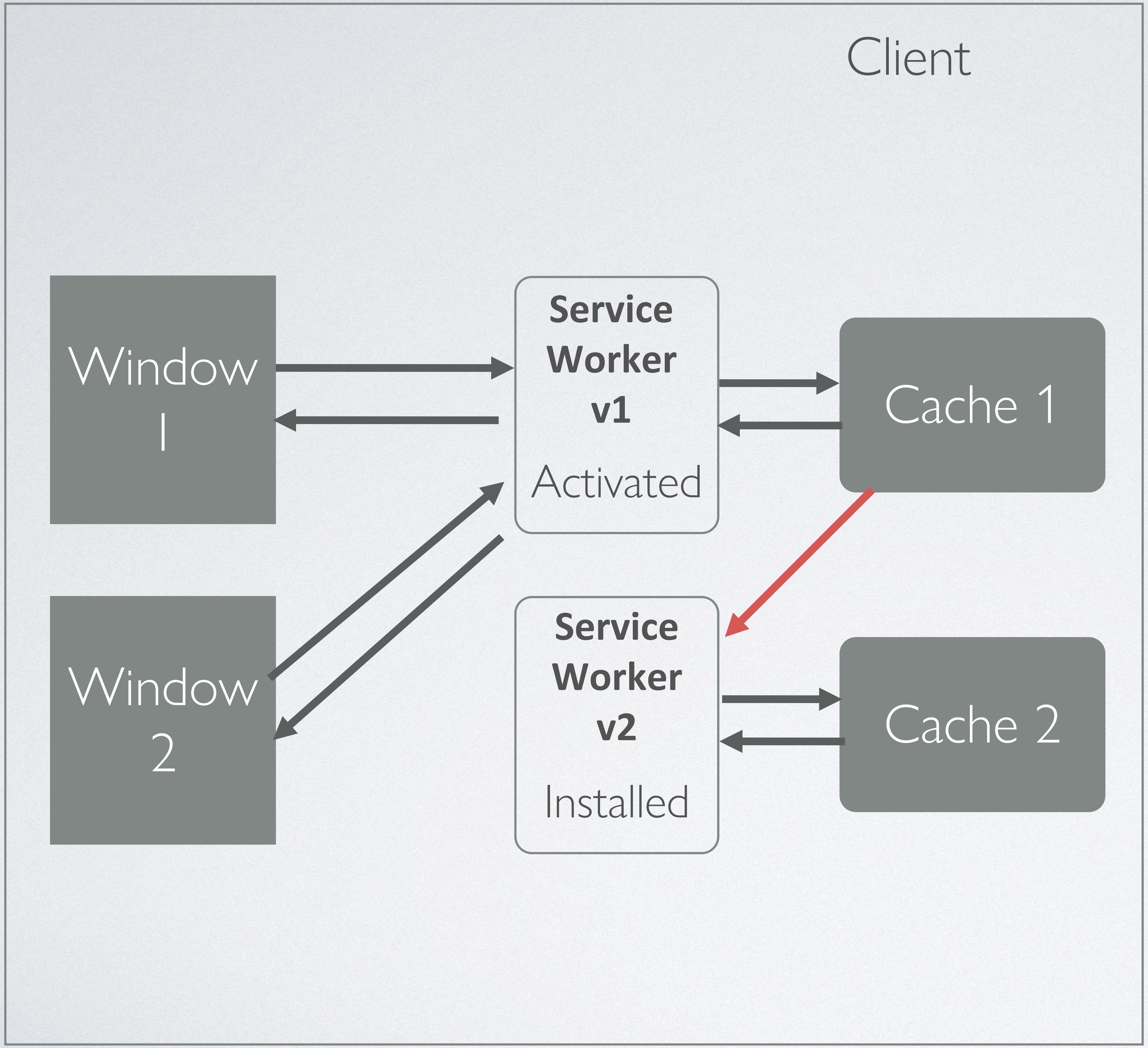
This is obviously not safe



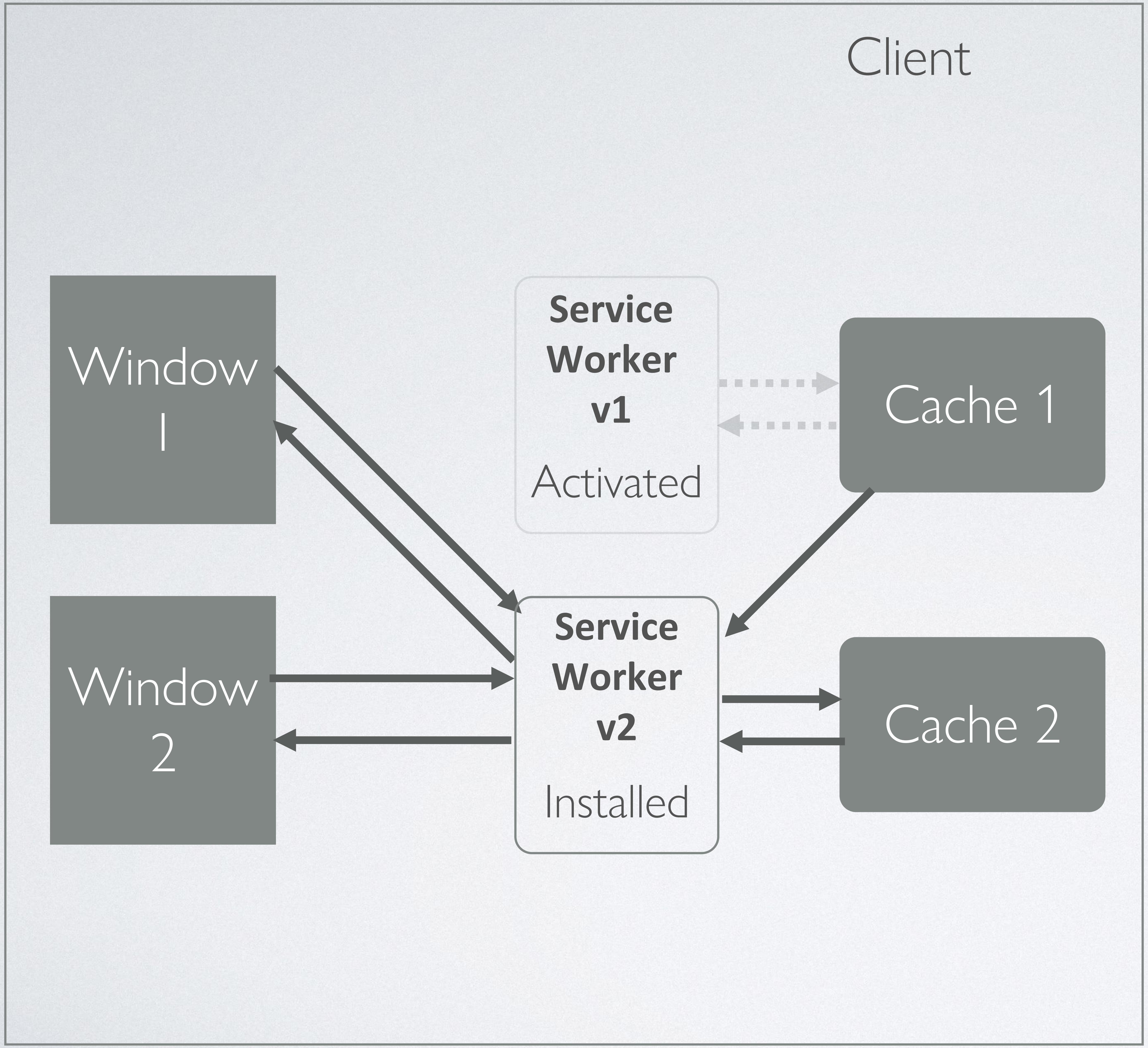
Instead the new service worker can **create a new cache** and set it up.

It still has access to the old cache but shouldn't try and change any of its data at this point.

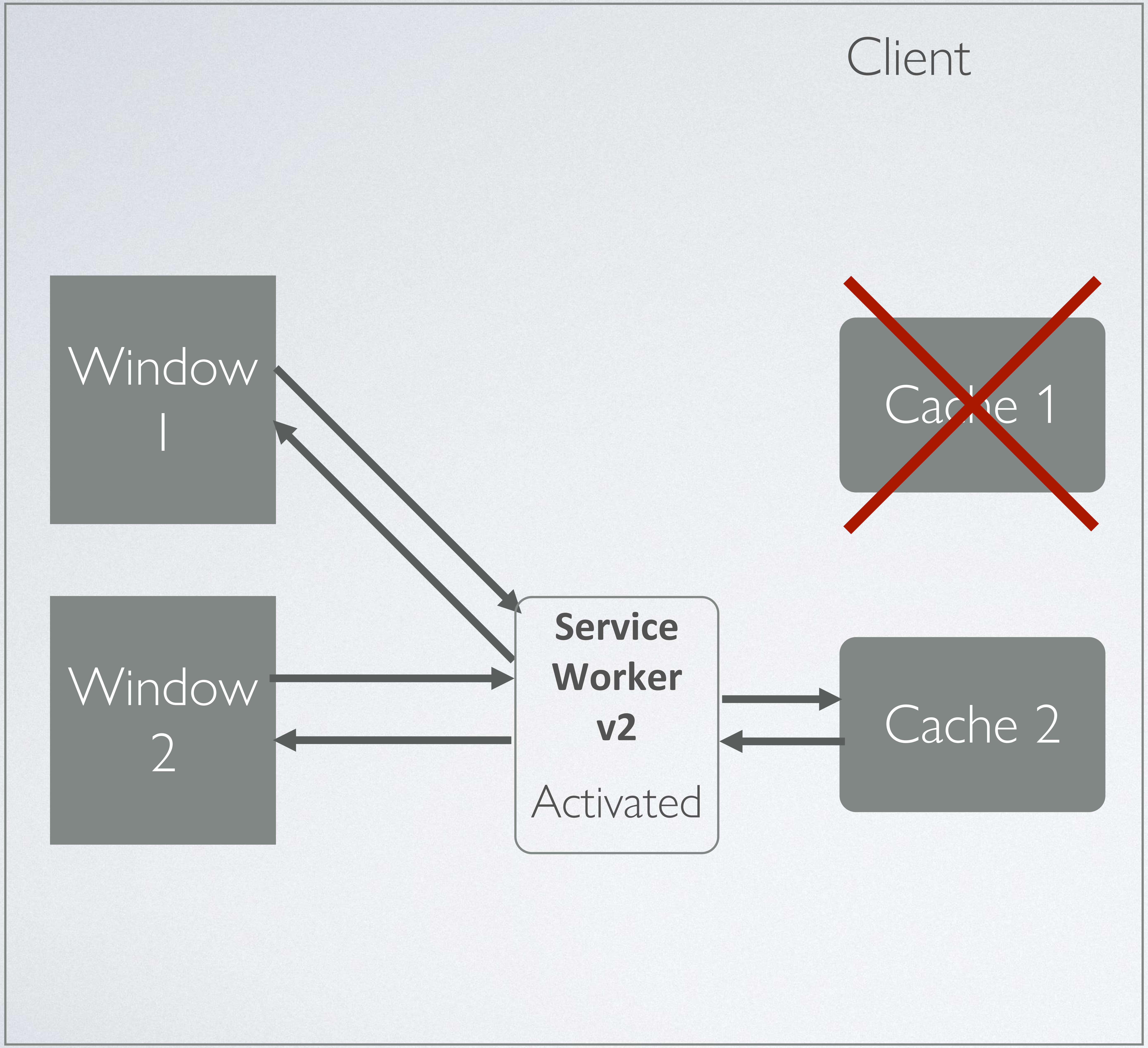
I.e. we usually just read from the original cache at this point (if we need it at all).



I.e. it is safe to copy items from the old cache to the new cache. But not to write to it (or delete from it).

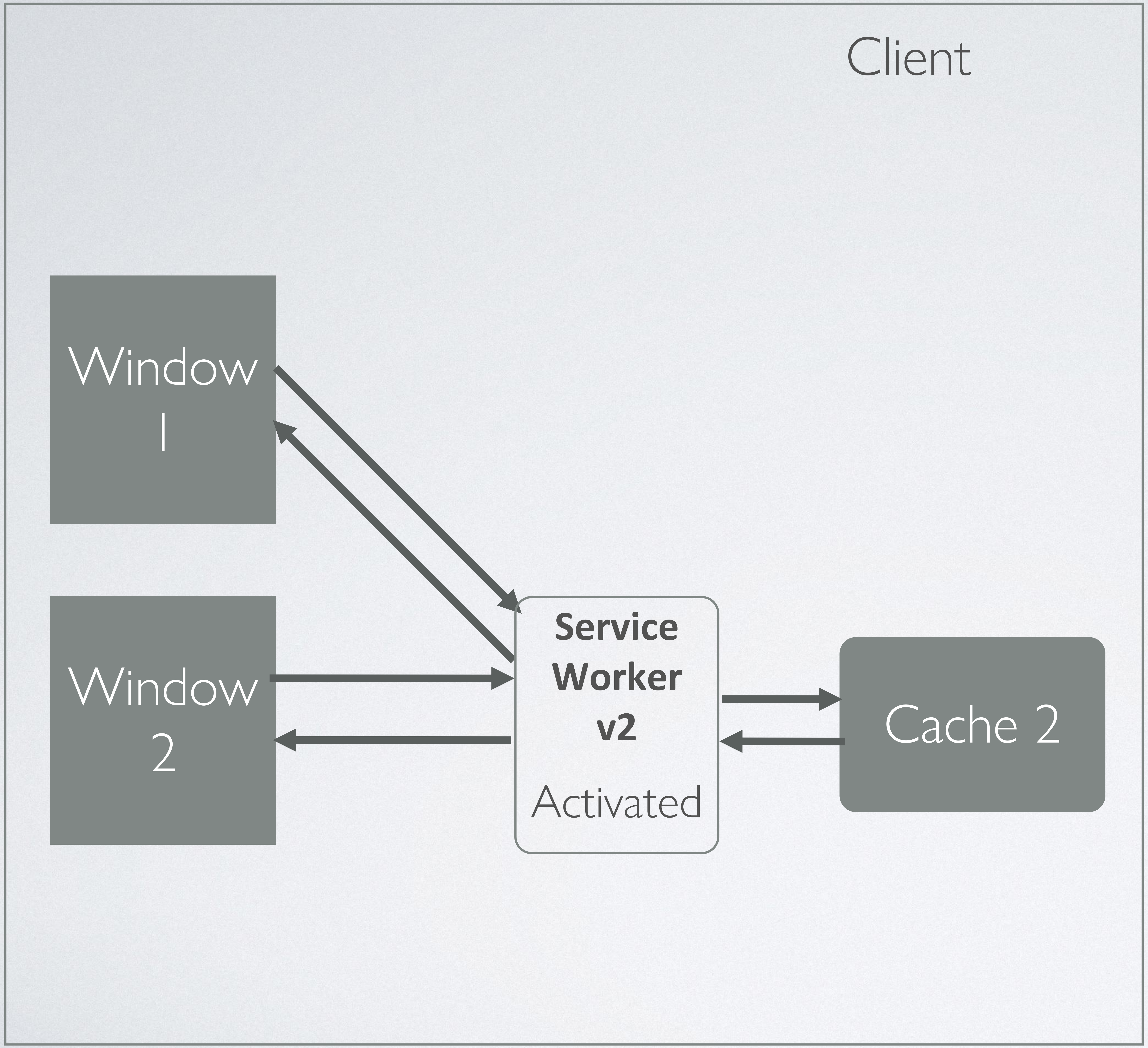


When all the windows/tabs start using the updated service worker it knows the old service worker (and its cache) are no longer in use.



I.e. once all the windows/tabs start using the new service worker, it enters the **activation phase**.

In this phase it can delete the old cache as it now knows the previous service worker is no longer using it.



We have now successfully updated the service worker.

To summarise, the two phases of installing a service worker are:

1) **Installation** (when you know there is probably another service worker in operation and this new service worker is not controlling any windows/tabs)

2) **Activation** (when you know all the windows are using this new service worker, and the previous service workers are no longer in use)

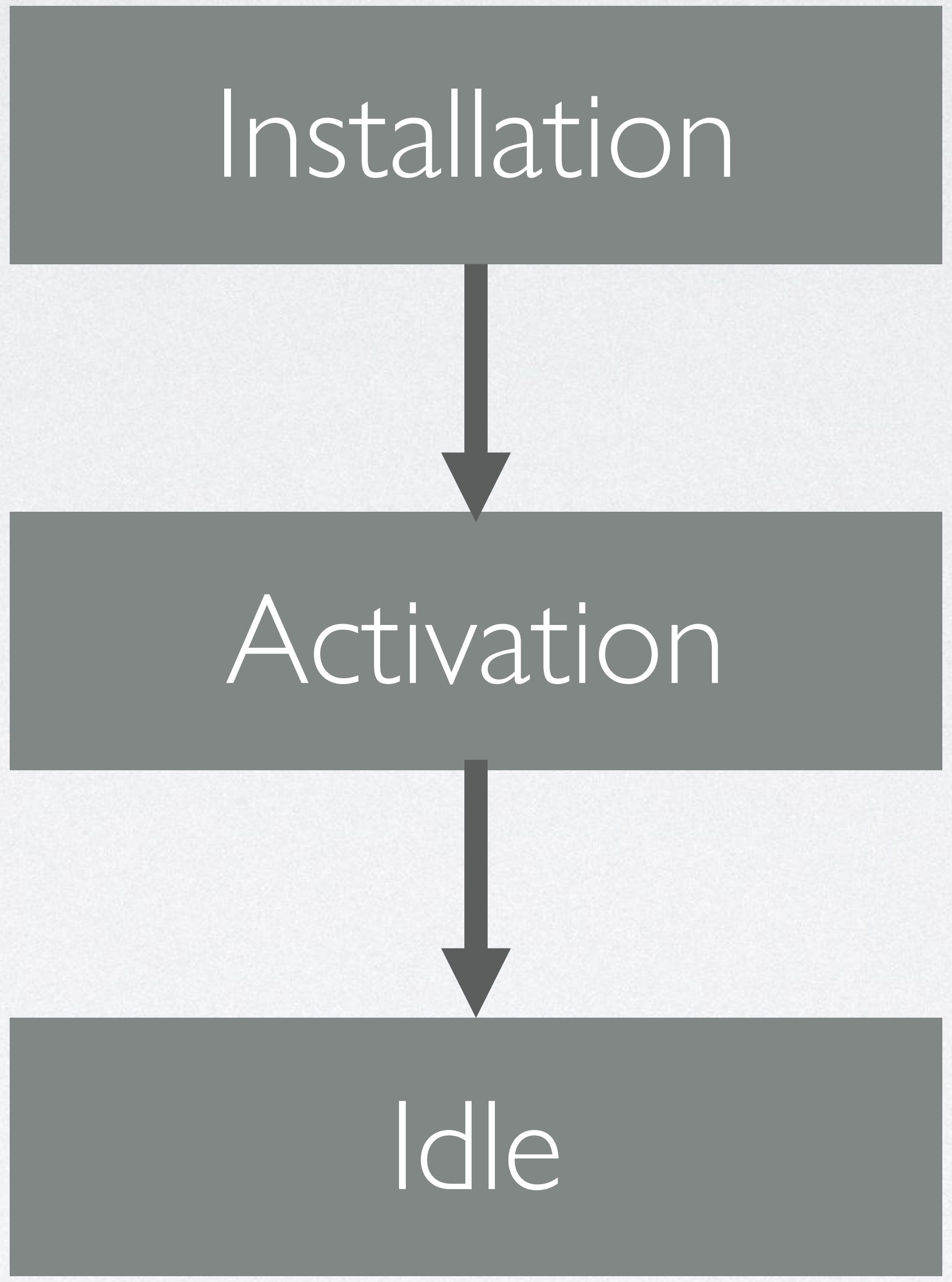
Note: If you load a page that downloads a new/updated service worker, the original service worker was used to download this newer version. Therefore the original service worker is still active and the new worker is not activated.

If you reload this page again (and no other pages are using it) the service worker should activate.

I.e. the benefits of an updated service worker will only be felt the next time you view the page ***at the earliest.***

Note: Even if the service worker file is in the cache, the slightest change to the original file on the server will cause it to be downloaded again.

Now that the service worker is activated it waits for fetch events from the windows/tabs that registered the service worker. This is the **Idle** Phase.

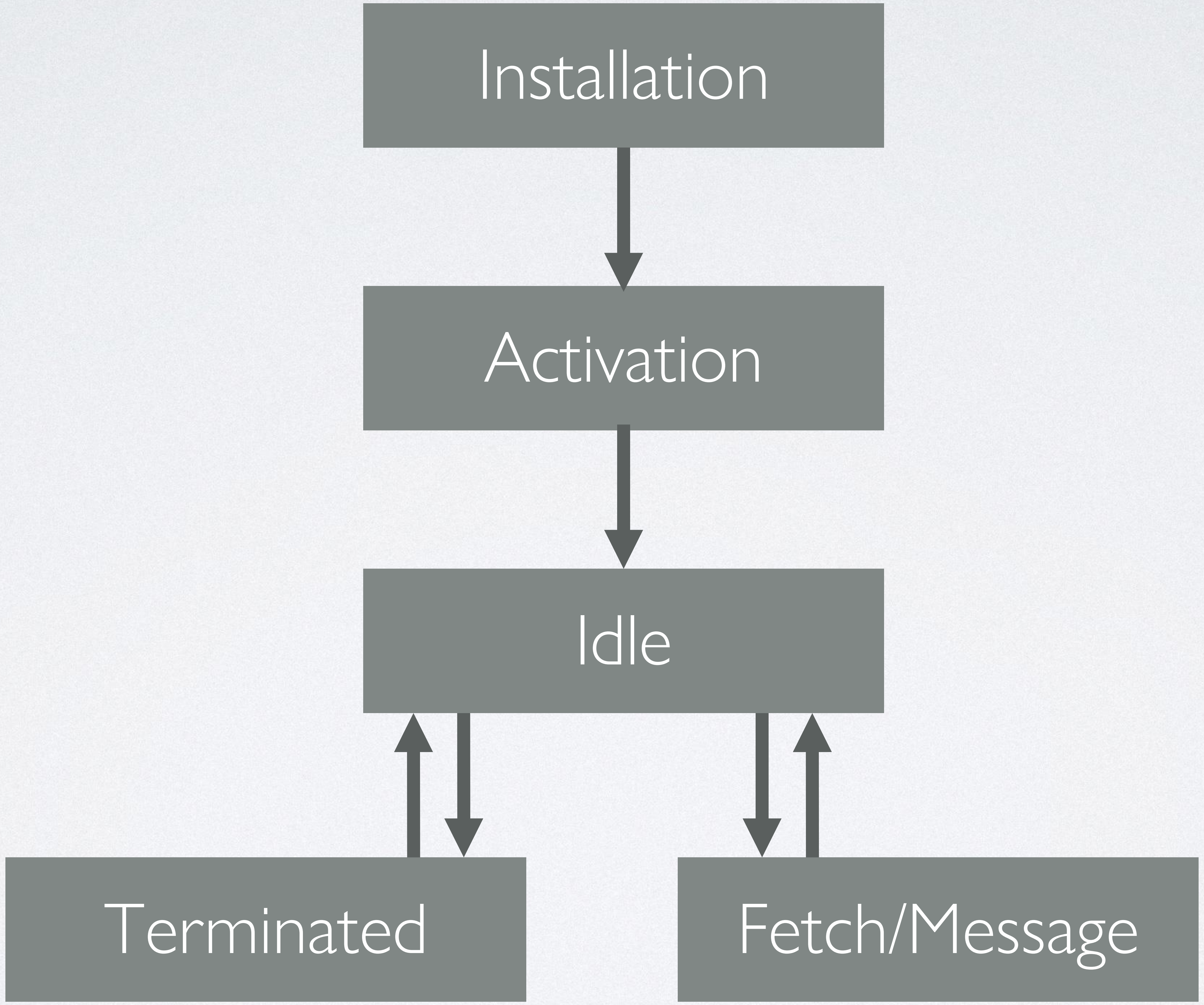


Installation

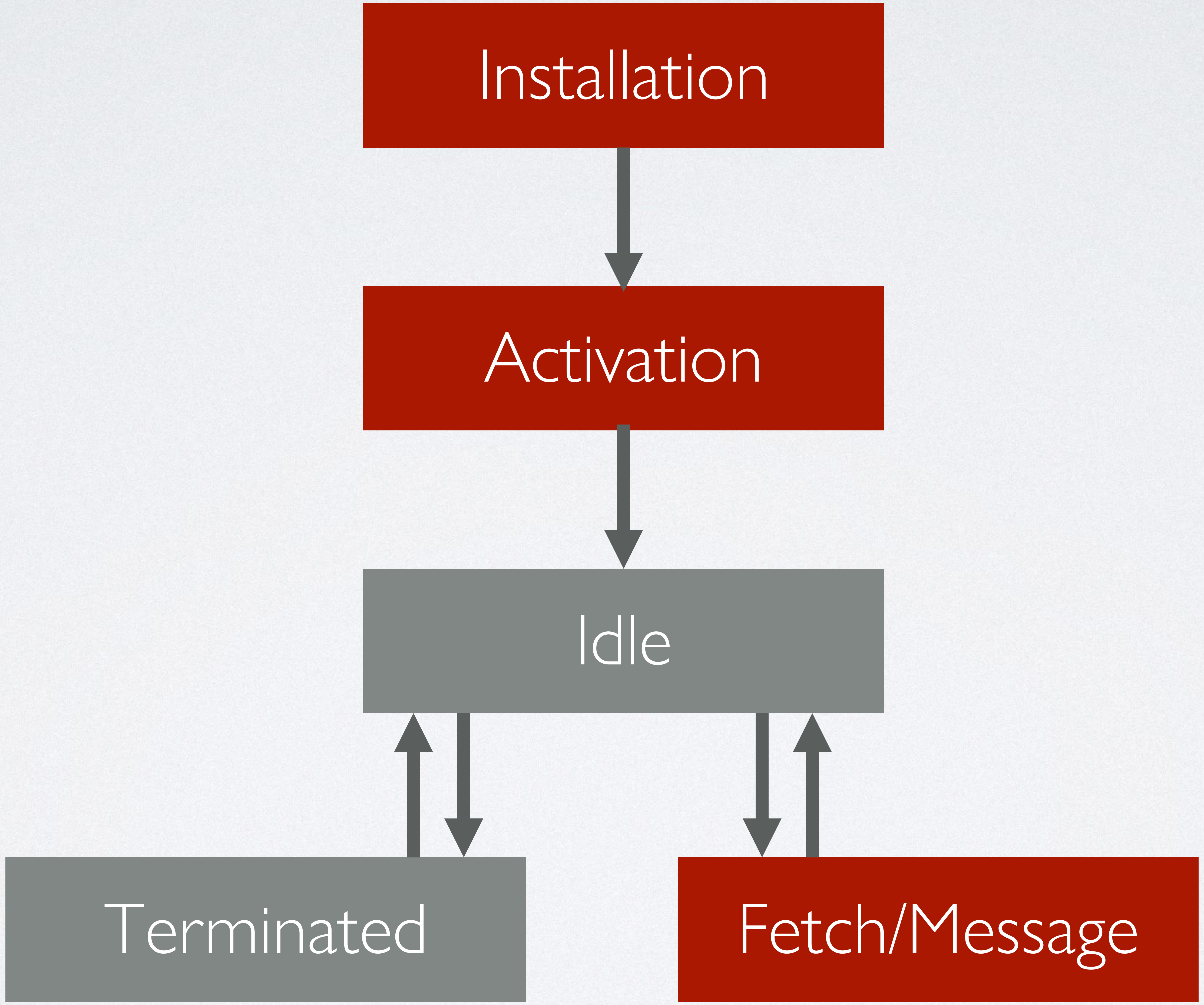
Activation

Idle

If a **fetch event** is received it will handle it. If not in use it may **terminate**. Note that a terminated service worker can be restarted so you can't assume a global state.



In our own service workers, we tend to write event handlers to detect **installation**, **activation** and **fetch events**.



Generally speaking the three phases can be used for the following:

Installation	Setting up a new Cache. Adding files to the cache (e.g. app shell assets)
Activation	Deleting old caches
Fetch	Deciding what to return to windows/tabs when they request a resource. Adding files to cache.

Timing

When you register a service worker in your code is up to you (since it won't be used as your service worker until, at least, the next time you load the page).

However, if you do it very early on the installation phase may start straightaway. This may have performance ramifications for your current page.

For example. you may want to delay registering a Service Worker until after some initial task. For example, you may want to delay the registration until after you run an initial animation so it doesn't interfere with its performance.

Note that, at best, a service worker will only be used the **next time** you load the page.

This means your page must be able to work without it.

This has the advantage of ensuring your page uses **Progressive Enhancement**. I.e. your application will have to work as a regular web page first, without any PWA enhancements.

Scope

The **scope** for a Service Worker refers to the web pages that it can serve.

Usually a service worker can be used by any web page in the same directory (or in a subdirectory).

It cannot be used by web pages in its parent directories.

For this reason we usually put service workers in the root directory of our app.

Once a service worker is registered is applies to all pages in its scope.

Inside the Service Worker

Events

Inside the service worker you can capture the three main events by adding event listeners to the global object.

```
self.addEventListener("install", function(e)
                                 { ......... });

self.addEventListener("activate", function(e)
                                 { ......... });

self.addEventListener("fetch", function(e)
                                 { ......... });
```

Those three events are:

Install: When the installation phase starts.

Activate: When the activation phase starts.

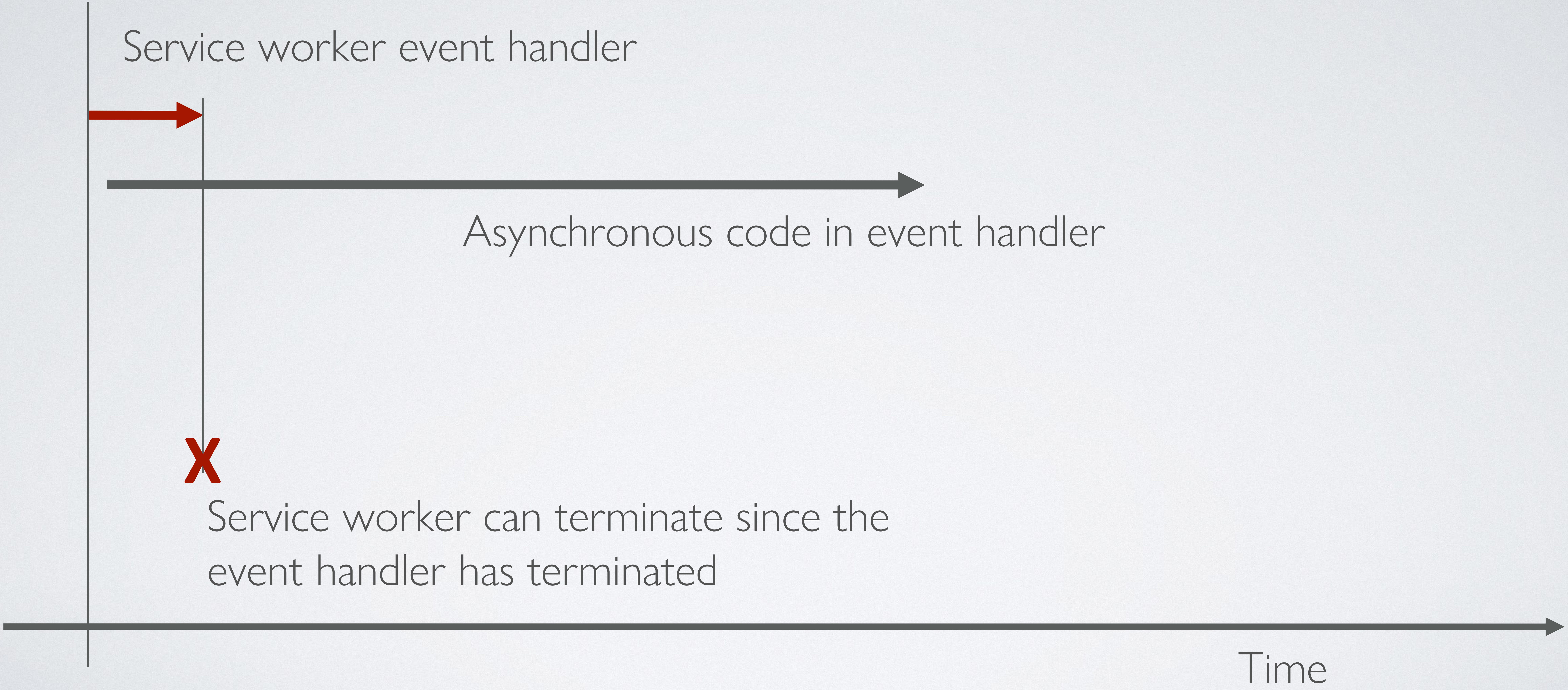
Fetch: This is fired whenever the web page requests some resource. This can be embedded media (e.g. <img> tags in the HTML), clicking on links, JavaScript code (E.g XMLHttpRequest objects, JSON-P, Fetch, etc), and so on.

Event Handlers & Service Worker Termination

If an event handler is triggered it can start up the service worker. The service worker can then terminate once the event handler has finished.

However, if the code in the event handler is asynchronous (e.g. it might start downloading resources over the internet) the event handler itself may be finished before the code it initiates is.

This can be problematic because if the event handler finishes before any asynchronous code then the service worker is free to terminate as well.



Service worker event handler

Asynchronous code in event handler

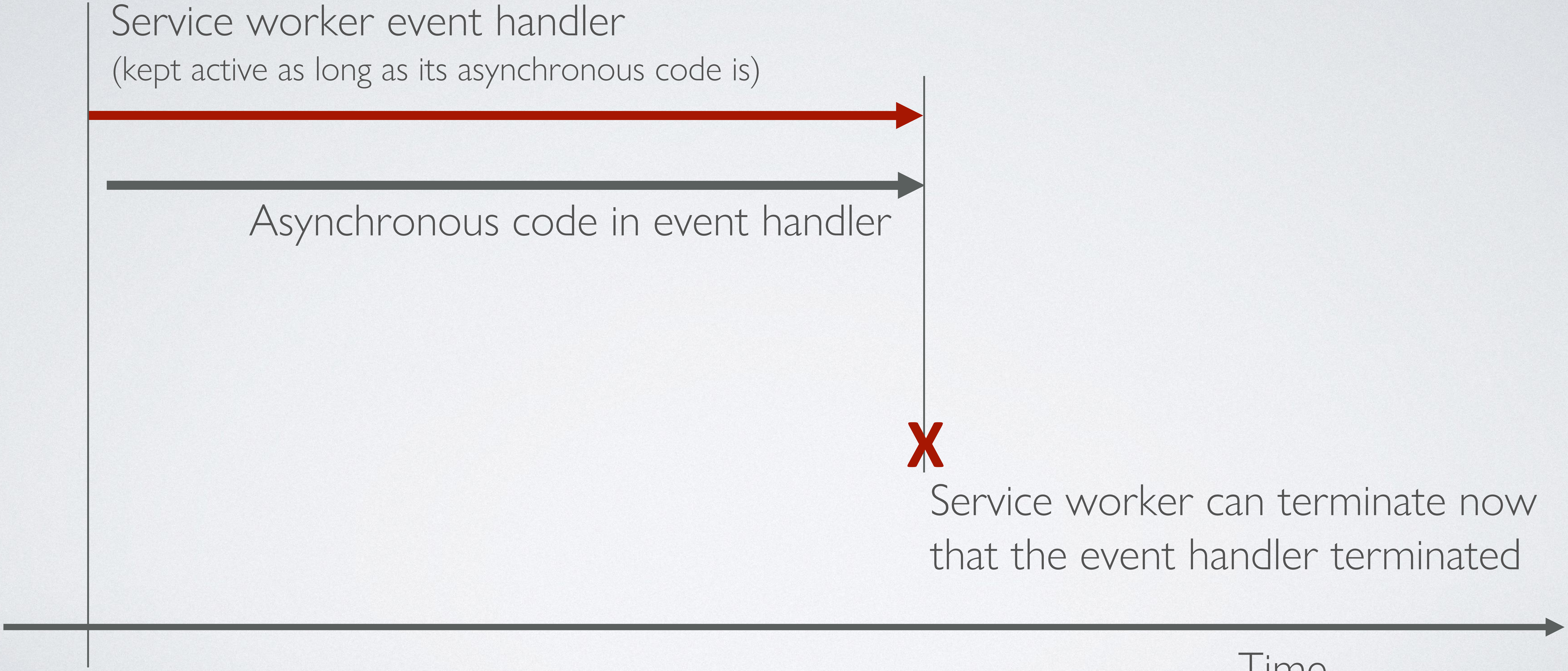
Service worker can terminate since the
event handler has terminated

Time

To avoid this we make the event handler take as long as its asynchronous code to execute.

We can use the **waitUntil()** event method to make sure the service worker knows the process is still underway. This is a method of the **Event** object that is passed to the event handler. It is passed a Promise and won't terminate until the Promise is resolved/rejected. (The Promise should resolve when the asynchronous code of the event handler is finished.)

```
self.addEventListener("activate",
                      function(e)
                      {
                       e.waitUntil( Code that returns a Promise )
                      });
```



Service worker event handler

(kept active as long as its asynchronous code is)

Asynchronous code in event handler

X

Service worker can terminate now that the event handler terminated

Time

Generally speaking the **install** event will copy files into a new cache, and the **activate** event handler can delete the previous cache. These are each invoked once.

The **fetch** event handler is called every time your web pages requests a resource over the internet.

In the **fetch** event handler (assuming, in this example, that it is passed the fetch event object in the **event** parameter) you must call the **event.respondWith()** method to pass a **Response** object back to the window or tab that made the request. The Response object represent the resource that was requested.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch("testfile.html"));
});
```

You must pass the function a **Promise** that resolves to a **Response** object.

In the service worker you write the code that decides what is sent back to the app when they make a request.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch("testfile.html"));
});
```

fetch() returns a **Promise** that resolves to a **Response** object for the "testfile.html" request. I.e. it will try and load the file from the internet and create a Response object with what is returned.

As we will see. the APIs we use in service workers make heavy use of Promises, **Response** objects and **Request** objects.

Cache API

We can create a cache by giving it a name. We can use the built-in instance of **CacheStorage** (called **caches**) to create a cache.

This command will open an existing cache with the name you provide, or, if it can't find it, it will create a new cache. It returns a **Promise** that resolves to a **Cache** object. This **Cache** object allows you to control the cache.

```
caches.open("myCacheName")
```

As mentioned, the **caches.open()** method returns a promise that resolves to a **Cache** object.

```
caches.open("myCacheName")
               .then(function(myCache)
                 {

                  Code that does something with myCache

               });
```

The **Cache** object representing the **"myCacheName"** cache will be copied to this parameter

```
caches.open("myCacheName")
               .then(function(myCache)
                {
```

Code that does something with *myCache*

```
               });
```

Now that you have an object that represents a cache you can use the **Cache** API.:

In the following examples, assume we have a **Cache** object called ***myCache***. We can manipulate the cache by using its API methods.

E.g.

```
myCache.put(req, resp);
```

This puts the file/resource (as a **Response** object) into the cache mapped to a corresponding **Request** object.

```
myCache.put(req, resp);
```

This is a **Request** object.

E.g.

```
var req = new Request("index.html");
```

```
myCache.put(req, resp);
```

This is a **Response** object.

E.g.

```
fetch(req).then(function(resp){
            myCache.put(req, resp)
})
```

Remember that when we are accessing resources we use two main type of object.

Request objects represents the URL of some resource. Many of the methods requiring a Request object as a parameter will accept the URL as a string (they will create the Request object from it). These objects can identify resources online or in the cache.

Response objects represent the data/resource that come back from a request (they can come from the network or the cache)

```
myCache.add(req);

myCache.add("arrow.jpg");
```

This puts the provided resource into the cache.

The parameter should be a **Request** object (**or the URL as a string**).

add() will retrieve the corresponding **Response** object for the **Request** object you passed to it.

It also differs from **put()** in that it checks that the response that is returned is valid (status is 200).

```
myCache.addAll([req1, req2, req3]);

myCache.addAll(["index.html", "icon.png"]);
```

This puts the provided resources into the cache.

The parameter should be an array of **Request** objects (or strings of URLs).

```
myCache.delete(req);

myCache.delete("index.html");
```

This deletes the provided request from the cache.

The parameter should be a **Request** object (or a string of the URL).

It resolves to **true** if it succeeds or **false** otherwise.

```
myCache.match(req);

myCache.match("index.html");
```

This checks if the provided request is in the cache.

The parameter should be a **Request** object (or a string of the URL).

It resolves to the first **Response** object that matches the request (or **undefined** otherwise).

```
myCache.matchAll(req);

myCache.matchAll(".jpg");
```

This checks if the provided request is in the cache.

The parameter should be a **Request** object (or a string of the URL).

It resolves to an array of **Response** objects that matches the request (or **undefined** otherwise).

E.g. Find ***all*** the files containing ",pdf" in the cache.

An array of all the matches

```
myCache.matchAll(".pdf").then(function(respArr) {
    console.log(respArr[0]);
});
```

Access one of the **Response** objects in the array.

The following example deletes all the files in the **/images/** folder from the cache.

Note the extensive use of Promises.

Every action is asynchronous (opening the cache, finding the files, deleting the files).

This is an example of deleting alll files in the **/images/** folder from the **example-cache** cache. In this example (from Google) the cache will be in the **cache** variable/parameter.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

Opening the cache (with the name 'example-cache') returns a promise that resolves to a Cache object (which we pass to the ***cache*** parameter below). We use the built-in **caches** object to open this cache.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

We take this **Cache** object (**cache**) and find all the **Response** objects in it that have the given string in their URL. The **matchAll()** function returns a Promise that resolves with an **array** of the matching Response objects.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

We then loop through this array (with the **forEach** array method) passing each individual **Response** object to the anonymous function.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

Using the **Cache** object (**cache**, which is still in scope) we delete each of these **Response** objects from the cache.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

The **delete()** method below is also asynchronous and returns a Promise. But since we don't require some action to occur immediately afterward we don't need to use this Promise.

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
})
```

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

Note that in the last example we used the **Cache** object representing the 'example-cache' cache to search for matches. If you don't want to restrict your search to a specific cache, you can also use the global **Cache** object (**caches**) directly.

```
caches.match(".html").then ( ..... )
```

I.e. this first technique searches a named cache for a resource and returns a **Promise** that resolves to a **Response** object.

```
caches.open("myCache").then(function(c)
        {return c.match("polaroid-64.png")});
```

Whereas, this second technique searches the global **Cache** object (which also returns a **Promise** that resolves to a **Response**)

```
caches.match("polaroid-64.png");
```

Caching Policies

One of the purposes of the 'fetch' event handler is to implement your **caching policy**.

This is how you want your PWA to implement requests for network resources.

Every request for a resource comes to the service worker. Nothing happens in response to that request unless you explicitly code it in that event handler. Even simply returning the file that was requested has to be explicitly coded.

A **caching policy** specifies when you access content in the cache. It also specifies when you access content on the network. It also decides which files go in the cache in the first place.

Remember that files in the cache have 2 main advantages,

Files in the cache load far more quickly than the same files online.

Files in the cache are available even when the browser is offline.

Therefore a caching policy can do two things:

Speed up your application (by loading files from the cache rather than online).

Allow your application work offline.

And its not just a case of choosing either the cache or the network. It can be about prioritising them. I.e. you can try the network first but if that fails try the cache. Or vice versa.

The following examples assume the app shell files are in the cache (they would be added in the **installation** phase).

Network first policy

Example I:

When a HTML page is requested by your app:

Try to find it **on the network**.

If it **is on the network** return it to the requesting page

If it **is not on the network** look for it in the **cache**.

If it is in the **cache** return the **cached** version

If it isn't in the **cache** return a special page (that is in the cache, or that you create in the service worker) that contains an error message alerting the use to the fact that the site is offline.

Cache first policy

Example II:

When a HTML page is requested by your app:

Try to find it **in the cache**.

- If it is **in the cache** return that to the page/app
- If it is **not in the cache** look for it **online (i.e. on the network)**.
    - If it is **on the network** return the **network** version
    - If it isn't **on the network** return a special page (that is in the cache) that contains an error message.

You can also have different policies for different files. For example, assume you have a news app that displays new items daily.

The files that make up your app shell could be retrieved from the cache first to speed up the loading time (since they are unlikely to have changed).

Whereas you could look for the pages containing news items on the network first since they change more frequently.

Caching Files

You must also decide ***when*** (and ***which***) items get added to the cache. (And also decide if/when files get deleted from the cache.)

For example, since you know what they are you can add the files that make up the app shell to the cache in the installation phase of the service worker.

However, other files may be unknown to you at the time of writing the service worker. E.g. the news app mentioned previously would be downloading images to go with each new story. You could arrange to store every image you retrieved online to the cache to speed up subsequent requests.

The following example puts any files we come across that are not already in the cache into it in order to speed up retrieval the next time the user requests that file.

Cache first policy with caching

Example III:

When a HTML page is requested by your app:

Try to find it **in the cache**.

  If it is **in the cache** return that to the page/app

  If it is **not in the cache** look for it **on the network**.

    If it is **on the network** return the **network** version **and place it in the cache**

    If it isn't **on the network** return a special page (that is in the cache) that contains an error message.

This last version means the cache is being is being filled up as we browse the web, so we might also want to delete files from it occasionally

Case Studies

The following examples are from different sources and illustrate the flexibility of service workers.

In some cases they can show several different techniques for achieving the same functionality.

You can follow the included links for more detailed information.

In the examples that follow, statements like the one below return the first value (**responseFromCache**) if it evaluates to a truthy value, otherwise it evaluates and returns the second (**responseFromFetch**).

```
return responseFromCache || responseFromFetch;
```

If the second part is functioning code or an expression (rather than just a value) then it is only executed or evaluated **if the first part is falsy**.

```
return responseFromCache || getNetworkResponse();
```

It is the equivalent of:

```
var temp = responseFromCache;

if (temp)
    return temp;
else
    return getNetworkResponse();
```

Case Study: **Install Event Handler**

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

The following code caches the files needed for the app shell.

```
var cacheName = "test":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

```
var cacheName = "test1":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

We store the cache name conveniently here. Whenever we update the Service Worker we can rename the cache here so it doesn't try and use the same cache as the previous version of the Service Worker.

Code to add known files to the cache is usually executed when the service worker is being installed.

```
var cacheName = "test":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

```
var cacheName = "test":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

We want to add this array of filenames to the cache.

```
var cacheName = "test";
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

We open the cache "test" which returns a Promise that resolves with the corresponding **Cache** object which we then add the files to.

```
var cacheName = "test":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

We must return the **Promise** we receive from **cache.addALL()** since the **event.awayUntil() function** requires a **Promise**.

Now the event handler will be blocked until the **addAll()** method has finished (i.e. the Promise it returns is resolved).

```
var cacheName = "test":
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

Case Study: **Activate Event Handler**

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers#Updating_your_service_worker

In the **activate** event handler you can delete previous caches to prevent the cache filling up.

In this example, we delete all caches except for those whose names appears in the ***cacheWhiteList*** array.

(**caches.keys()** returns a Promise that resolves to an array of the names of all the caches. And the **map()** method applies the same function to all the elements of an array.)

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

This code is executed when the service worker is activated.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

We want to delete all the caches except the ones whose name appear in this array.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

caches.keys() returns a Promise that resolves to an array of all the cache names in **caches**.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

The **map()** array function is passed a function as an argument to which it passes each individual element of the receiving array (**keyList**). Each invocation of the function returns a value (a Promise in this case) which are collected in an array and returned by **map()**.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);

      }));
    })
  );
});
```

We check if the current cache name (in **key**) is in the white list array.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

I.e. we delete each database (excluding the ones in **cacheWhiteList**). Each deletion returns a Promise. These are collected and returned as an array by **keyList.map()**.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

We can then wait until every Promise in that array has resolved.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['v2'];

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

Expects Promise

Promise.all() returns a promise...

... that resolves only when all these Promises resolve

The following variation works with our previous "install" handler example that stored all its resources in a cache whose name was stored in the **cacheName** variable (i.e. we use a single variable instead of an array)

Now this "activate" handler deletes all caches except the one stored in **cacheName**.

```
var cacheName = "test":

self.addEventListener('activate', function(event) {

  event.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (key != cacheName) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

Case Study: Fetch Event Handler - Check Cache Only

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

The general format for a fetch event handler is the following:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(

      Code that returns a Promise that resolves to a  Response object

     ));
});
```

We specify that the event handler is for every network request coming from the pages that registered the service worker (i.e. **fetch** events)

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
```

Code that returns a Promise that resolves to a **Response** object

```
      ));
});
```

The actual **Request** object representing the file that is being requested is available via the event argument that is passed to the event handler (via its **request** property)

I.e. **`event.request`**

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
```

Code that returns a Promise that resolves to a **Response** object

```
      ));
});
```

The **respondWith()** method of the **Event** object must be passed a Promise that resolves to a **Response** object. It then returns this **Response** to the page that made the request.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
     Code that returns a Promise that resolves to a  Response object
     ));
});
```

Therefore, the purpose of this code is to return a **Response** object corresponding to the requested resource *in a Promise*. Deciding whether that **Response** object comes from the network or the cache will be the main purpose of the code.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(

      Code that returns a Promise that resolves to a  Response object

      ));
});
```

In this example the event handler searches the cache and returns a Promise that resolves to the **Response** object it found. This means the web page will never see content from the network, only from the cache.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request));
});
```

Note that a lot of the functions we use (like **respondWith()**) require Promises that resolve to **Response** objects.

However, we rarely have to create these Promises ourselves as methods such as **match()** (for caches) and **fetch()** (for network requests) will automatically return these Promises.

Case Study: Check Network Only

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

This event handler effectively performs the normal function we would expect of our web pages. Any request object received by the handler is automatically used to make a network request (via **fetch()**).

And the Promise we get back from **fetch()** (which resolves to the corresponding **Response** object) is then passed to **respondWith()** (to go back to the requesting page).

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch(event.request));
});
```

Case Study: Check Cache, fallback to Network

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

This next event handler will check the cache first, but if it can't match the request in the cache it will retrieve the file from the network.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);}
    )
  );
});
```

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);}
    )
  );
});
```

match() returns a Promise which will resolve to the matching **Response** object from the cache...

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);}
    )
  );
});
```

... we can then return this Promise.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);}
    )
  );
});
```

However **match()** resolves to **undefined** if there is no match.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);}
    )
  );
});
```

If response is **undefined** (i.e. falsy) we return the result of the **fetch()** instead

Here is similar function from MDN that provides the same functionality but uses **async** functions and **await** instead of Promises and **then()**.

```
addEventListener('fetch', event => {
  // Prevent the default, and handle the request ourselves.
  event.respondWith(async function() {
    // Try to get the response from a cache.
    const cachedResponse = await caches.match(event.request);
    // Return it if we found one.
    if (cachedResponse) return cachedResponse;
    // If we didn't find a match in the cache, use the network.
    return fetch(event.request);
  }());
});
```

https://developer.mozilla.org/en-US/docs/Web/API/FetchEvent/respondWith

Case Study: Check Cache First, Then Network (And Cache the Response)

https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

This is similar to the earlier example with the added feature of caching any Responses that we retrieve from the network.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

If we have to retrieve the resource over the network....

...we pass the response here when it is ready.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

We add the Response to the cache ...

...associated with the original request

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

We return the response so it will be passed to **respondWith()**

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

Remember that when we return the Response from the function that we passed to **then()** that the **then()** function will automatically return a Promise that resolves to this value. And it is this Promise that is eventually passed to **respondWith()**.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

Note use of the cloned response as well as the original since the original's body can only be read once.

Case Study: Network First, Then Cache

-
https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker

Here we try to find the Request on the network first. If it is found we return the Promise returned by **fetch()**.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

If the Promise returned by **fetch()** is rejected (i.e. the file can't be retrieved over the network), we return the cached version instead (via the **catch()** Promise function.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

Case Study: Network First, Then Cache

https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage/open

The following code demonstrates the techniques we looked at previously used in a slightly different way.

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(event.request).then(function (response) {

        // response may be used only once
        // we need to save clone to put one copy in cache
        // and serve second one

        let responseClone = response.clone();

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(event.request).then(function (response) {

        //
        //
        //

        let responseClone = response.clone();

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

event.respondWith(caches.match(event.request).then(function(response) {

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(event.request).then(

        // response may be used only on
        // we need to save clone to put
        // and serve second one

        let responseClone = response.cl

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
if (response !== undefined) {

        return response;

      }
```

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fe

        // resp
        // we n
        // and

        let res

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
else {

         return fetch(event.request).then(function (response) {
```

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(event.request).then(function (response) {

        // response may be
        // we need to save
        // and serve secon

        let responseClone = response.clone();

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
let responseClone = response.clone();
```

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(even

        // response may
        // we need to s
        // and serve se

        let responseClo

        caches.open('v1

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });
```

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      return fetch(event.request).then(function (response) {

        // response may be used only once
        // we need to save clone to put one
        // and serve second one

        let responseClone = response.clone();

        caches.open('v1').then(function (cache) {

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

return response;

```
self.addEventListener('fetch', function(event) {

  event.respondWith(caches.match(event.request).then(function(response) {

    // caches.match() always resolves
    // but in case of success response will have value

    if (response !== undefined) {

      return response;

    } else {

      retur

        //
        //
        //

        let

        cac

          cache.put(event.request, responseClone);

        });

        return response;

      }).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
    }
  }));
```

```
}).catch(function () {

        return caches.match('/sw-test/gallery/myLittleVader.jpg');

      });
```

This image is returned if the request can't be found on the network or in the cache.

Case Study: Create and Return a Response in the Service Worker

It is possible to create the response you want to return in the service worker itself.

This technique is usually used for error messages. E.g. you might want to return a custom web page containing an error message if you can't reach a page online.

Return HTML

Here we return HTML. Note that we must specify the type of data we are returning in the custom Response object that we create.

```
self.addEventListener('fetch', function(event) {

var fetchedP = fetch(event.request);

return fetchedP
       .then(function(resp){return resp})

       .catch(function() {return  new Response(
                                      "<h1>Offline</h1>",
                                     {headers: {"Content-Type": "text/html"}}
                                  );
});
```

Return JSON

Here we return JSON (e.g. if our web page is accessing web services).

```
self.addEventListener('fetch', function(event) {

var fetchedP = fetch(event.request);

return fetchedP
      .then(function(resp){return resp})

      .catch(function() {return  new Response(
                                 "processData({status: false})",
                                {headers: {"Content-Type": "text/javascript"}}
                              );
});
```

Case Study: Minimal Service Worker

The following is a minimal service worker proposed by Jeremy Keith

https://adactio.com/journal/13540

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

1 of 2

```
const cacheName = 'files';
const offlinePage = '/offline/index.html';

addEventListener('install', installEvent => {
  skipWaiting();
  installEvent.waitUntil(
    caches.open(cacheName)
    .then( cache => {
      return cache.add(offlinePage);
    })
  );
});

addEventListener('activate', activateEvent => {
  clients.claim();
});
```

These functions force the web pages using the previous version of this service worker to use this new one (instead of waiting for the normal life cycle of a Service Worker to play out).

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

2 of 2

```
addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        const responseFromCache = await caches.match(request);
        return responseFromCache || caches.match(offlinePage);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```

You could also reduce this service worker to just the fetch event handler.

I.e. It doesn't have an install or activate event handler, and doesn't implement the offline page.

```
// HTML files: try the network first, then the cache.
// Other files: try the cache first, then the network.
// Both: cache a fresh version if possible.
// (beware: the cache will grow and grow; there's no cleanup)

const cacheName = 'files';

addEventListener('fetch',  fetchEvent => {
  const request = fetchEvent.request;
  if (request.method !== 'GET') {
    return;
  }
  fetchEvent.respondWith(async function() {
    const responseFromFetch = fetch(request);
    fetchEvent.waitUntil(async function() {
      const responseCopy = (await responseFromFetch).clone();
      const myCache = await caches.open(cacheName);
      await myCache.put(request, responseCopy);
    }());
    if (request.headers.get('Accept').includes('text/html')) {
      try {
        return await responseFromFetch;
      }
      catch(error) {
        return caches.match(request);
      }
    } else {
      const responseFromCache = await caches.match(request);
      return responseFromCache || responseFromFetch;
    }
  }());
});
```