Periodic Background Sync - Implementation Design

nator@
Contributors: beverloo@, rayankans@, jakearchibald@
Last modified: 01-14-2019
Status: [Draft | Under Review | Reviewed | Implemented | Obsolete]
This document is public.

Introduction

This document assumes that you have read the PRD.

Periodic Background Sync provides more background processing power to web apps, by allowing them to schedule a task to be run in the background periodically. The task will usually also require network connectivity, and will be used by apps to update state or content. Since network connectivity can be flaky, retry logic will also be necessary. (PRD)

IDL

The following IDLs can be added to accommodate Periodic Sync:

```
partial interface ServiceWorkerRegistration {
  readonly attribute PeriodicSyncManager periodicSync;
};

[Exposed=(Window,Worker)]
interface PeriodicSyncManager {
  Promise<void> register(DOMString tag, BackgroundSyncOptions options);

  Promise<void> unregister(DOMString tag);
  Promise<sequence<DOMString>> getTags();
};

partial interface ServiceWorkerGlobalScope {
  attribute EventHandler onPeriodicSync;
};

[Constructor(DOMString type, PeriodicSyncEventInit init), Exposed=ServiceWorker]
interface SyncEvent : ExtendableEvent {
  readonly attribute DOMString tag;
};

dictionary PeriodicSyncEventInit : ExtendableEventInit {
```

```
  required DOMString tag;
};
```

```
dictionary BackgroundSyncOptions {

  unsigned long long minInterval;

}
```

Implementation Goals

1. Allow a one-shot and a period sync registration to use the same tag.
2. Reuse one-shot Background Sync code as much as possible.

Implementation Design

Updates to Protos

BackgroundSyncRegistrationProto and BackgroundSyncRegistrationsProto can be extended thus: (changes in bold)

```
enum SyncPeriodicity {
  SYNC_ONE_SHOT = 0;
  SYNC_PERIODIC = 1;
}

message BackgroundSyncRegistrationProto {
  // required int64 id = 1;
  required string tag = 2;
  // optional SyncPeriodicity periodicity = 3;
  //  optional int64 min_period = 4;
  // required SyncNetworkState network_state = 5;
  // required SyncPowerState power_state = 6;
  required int32 num_attempts = 7;
  required int64 delay_until = 8;
}

message OneShotSyncRegistrationProto {
  optional BackgroundSyncRegistrationProto = 1;
```

```
}

message PeriodicSyncOptions {
  optional min_interval = 1;
}

message PeriodicSyncRegistrationProto {
  optional BackgroundSyncRegistrationProto = 1;
  optional PeriodicSyncOptions = 2;
}

message BackgroundSyncRegistrationsProto {
  repeated BackgroundSyncRegistrationProto registration = 1;
  // required int64 next_registration_id = 2;
  required string origin = 3;
}

message SyncRegistrationsProto {
 repeated OneShotSyncRegistrationProto one_shot_registrations = 1;
 repeated PeriodicSyncRegistrationProto periodic_registrations = 2;
 required string origin = 3;
}
```

Existing data on clients will need to be migrated, when the feature is enabled, to the new schema. This allows easy extensibility and makes any future changes only to One-Shot sync or only to Periodic Sync logic easy.

Waking up the browser

Non-Android

One-shot Sync doesn't keep Chrome running in the background if background syncs are registered. Periodic Sync will do the same.

Android

One-shot sync listens to network requests even when browser isn't running, and wakes the browser up if background syncs are registered. Periodic Sync will do the same, with two differences:

1. We'll cap the number of times per day we wake Chrome up for periodic Background Sync tasks. This cap will be Finch configurable, and will default to 2. Once the browser is running, any ready background sync events will be immediately fired.
2. The current implementation schedules a GCM event to schedule a task to wake the browser up when the device goes online. The recommended way to do so is to use JobScheduler Android L onwards. We will thus update to using BackgroundTaskScheduler to call the the appropriate class to schedule this task.

Scheduling the Wakeup Task

Recall that a Background Task using BackgroundTaskScheduler will be set up to wake Chrome up a few times a day, controlled by Finch, and defaulted to 2. This will be scheduled when the first time a new registration is made and there isn't a background task set up.

If at any point, there are no active_registrations, we cancel this task.

Integration with Site Engagement

We won't run periodic tasks for a site if the user engagement with the site is NONE. This info will help us create a list of suspended_registrations, which is a list of registered origins that the user hasn't engaged with in a while. Note that we don't simply delete these registrations because clearing of browsing history resets these scores.

Beyond that, the frequency of the periodic task will be in direct proportion to the user's engagement with the site.

Site engagement scores range from 0.0 to 1.0 and are divided into six EngagementLevels. Based on the six levels, we can assign a suggested site_engagement_frequency.
This will help us pick a delay_until for these sites, let's call it site_engagement_delay_until:

NONE - Suspended
MINIMAL - 4x hours
LOW - 3x hours
MEDIUM - 2x hours
HIGH - x hours
MAX - min_sync_recovery_time (or an additional parameter)
where x is the number of hours after which Chrome will be woken up to process Periodic sync registrations, which is currently set to 2. (see previous section)

NOTE: We will not be allowing any website to run code hourly; please see 'When to Fire' for details on how the site_engagement_delay_until will be used.

Listening to changes in site engagement scores is an options, but it has two drawbacks:

1. We'll be informed of changes in site engagement scores for all origins, most of which we won't care about.
2. Very frequent changes in scores of sites we care about will cause us to constantly change the next scheduled time, which is wasteful, and makes debuggability hard.

Therefore, it's better to consider the site engagement score for an origin when deciding when to schedule the next periodic task for a registration. At this time, we can mark this registration as active or suspended, and update its site_engagement_delay_until. Once the delay_until for a registration has been decided, it can be displayed to the developer via devtools and shouldn't be changed until the event has actually fired, and any retries attempted.

Whether a registration is active or suspended doesn't need to be persisted, since it's better to query the freshest site engagement scores when Chrome is brought up and update the list in memory.

Triggers

One-shot Sync events

One-shot Sync has a trigger-based mechanism for firing sync events. Assuming that Chrome is already running, the following triggers causes us to fire any ready events for One-shot Sync:

1. Changes in ServiceWorkerContext: a registration is deleted, or the storage partition is wiped out.
2. Changes in network state (on Android we listen for these changes even when the browser isn't running).
3. Background Sync Wake alarm fires. More on how and when this is scheduled later.

Whenever any of these triggers happen, we update the list of events ready to fire,* which is a subset of active_registrations. Once this list is ready, we fire each of these events, and for any that don't succeed and can be retried, we update the delay_until, and put them in the queue of events to be fired again.

* Any events with delay_until in the past are ready to be fired.

Periodic Sync events

Since the use cases and promises for periodic events are different, they don't have to be fired immediately. **We therefore err on the side of caution and only wake Chrome up a few times a day, which will be Finch configurable, and defaulted to 2**. This way, if Chrome is scheduled to be woken up, we always know the soonest time that will happen next.

On Android, Chrome will almost always not be available, having been reclaimed by the OS, and will have to be woken up to fire periodic sync events. Since Android is the more resource constrained of the target platforms, it makes sense to design the firing of events around it.

There's no need for scheduling a delayed task then, in addition to the background task that wakes Chrome up with network connectivity. The list of triggers that cause us to reschedule tasks, thus reduces to:

1. Some ServiceWorker Context changes -- a registration being deleted, or the storage being wiped down.
2. Changes to the periodic Background Sync registration -- caused by register() or unregister().
3. Chrome waking up.

Here's how we'll react to each trigger:

1. *ServiceWorker changes:* Delete all Periodic Background Sync registrations, and the Chrome wakeup background task.
2. *register() or unregister():* Add or remove from the list of registrations. If adding, set delay_until for the registration*, and potentially update the Chrome Wakeup Task.
3. *Chrome waking up:* Fire any events with delay_until in the past, retry any tasks that failed with an exponential delay, and set delay_until for all registrations*.

* We discuss how to do that next.

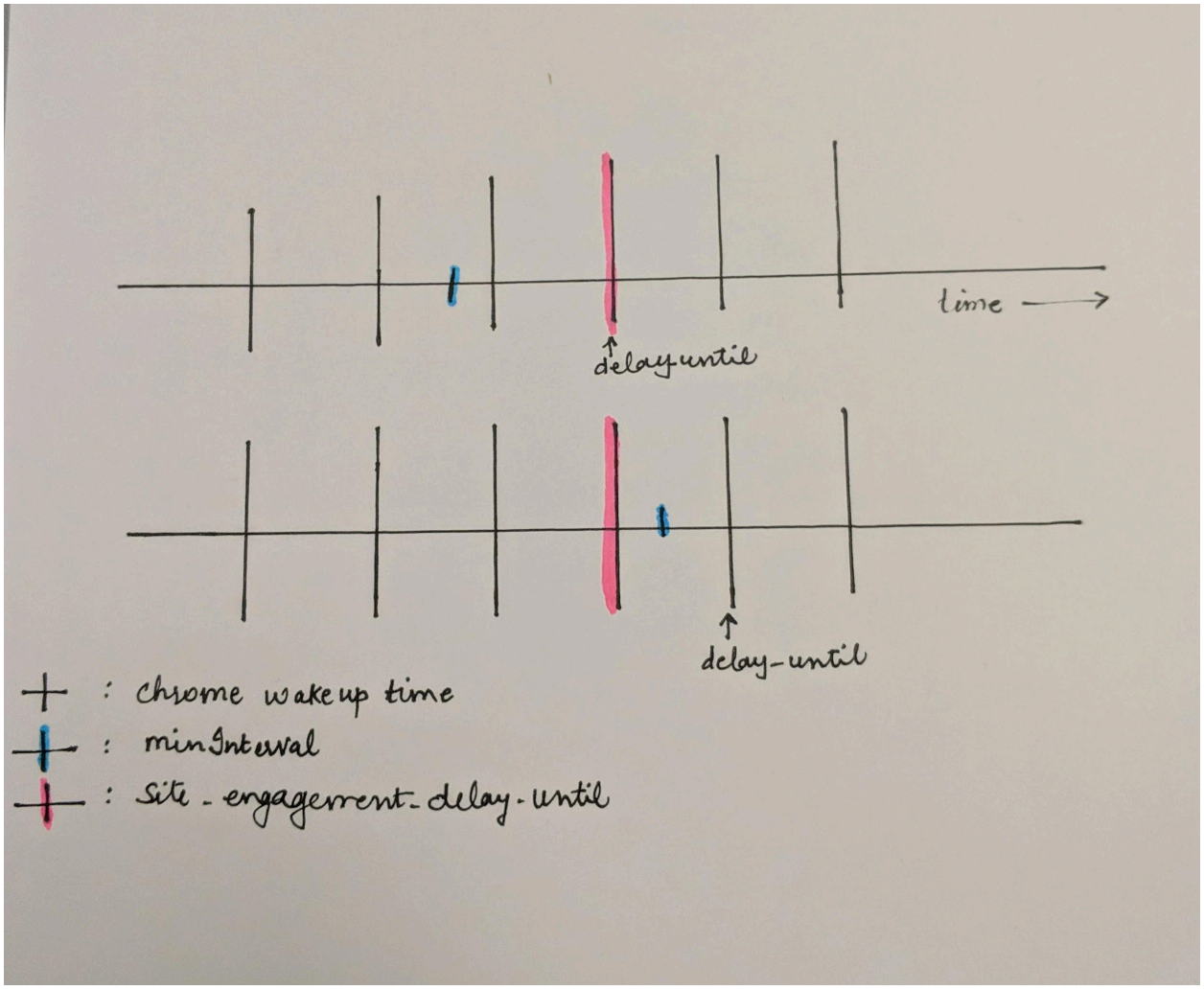
When to fire

If there is a task scheduled to wake Chrome up, let next_wakeup_time be the next time Chrome wakes up. Based on that and the period after which we'll wake Chrome up, we can determine each time Chrome will be woken up next.

For each of the registrations, we

a. Get the site engagement level for the origin. If it's NONE, suspend the registration (which means no periodic sync event will be fired for it). If not NONE, get a site_engagement_delay_until for the origin as described in Integration with Site Engagement. This will be a multiple of the number of hours we'll wake Chrome after.
b. If minInverval < site_engagement_delay_until, delay_until is the same as site_engagement_delay_until.
   If minInterval > site_engagement_delay_until, delay_until is the soonest wakeup time after minInterval + current_time.

Both are explained in the picture here, the horizontal line is time increasing to the right, and the vertical black lines are the times when Chrome is scheduled to wake up:



Retries

We allow two retries per task, if it was unsuccessful, with an exponentially increasing delay/backoff. Retries, if allowed, are scheduled immediately after a task is unsuccessful, but do not wake Chrome up.

A developer can abuse this system by always failing the onperiodicsync event to get three attempts per task. A temporary suspension logic can be implemented, where Chrome decides to skip the next N attempts if such an abuse is detected. Alternately, retry logic can be skipped until network detection on Android gets better.

Code structure

New IDL file and mojo interfaces will need to be written for Periodic Background Sync, as explained above. However, a lot of the implementation for one-shot Background Sync can be reused. A good way to allow both features to be developed independently while reusing shared logic is to have the following structure for many BackgroundSync classes:

```
class BackgroundSyncComponent {
  void methodA(arg1, arg2, … ); // contains shared logic.
};

class OneShotSyncComponent : public BackgroundSyncComponent {
  // methods with non-shared logic
  int DoTask_OneShotSync();
}

class PeriodicComponent : public BackgroundSyncComponent {
  // methods with non-shared logic
  int DoTask_PeriodicSync();
}
```

background_sync.mojom can be split like so:
Shared data structures stay in it.
One-shot sync specific structures and logic move to one_shot_sync.mojom, and
Those specific to Periodic sync are added to periodic_sync.mojom, both of which import background_sync.mojom.

Logic that can be reused

1. ~~Permission logic. Since we're reusing the Bakground Sync permission, we can reuse the same permission checking logic for periodic Background Sync.~~
2. Most of BackgroundSyncController, BackgroundSyncContext, with any updates listed in the next section.

Logic that's new or different

1. New IDLs and updates to background_sync.mojom.

2. New mojom files and their implementations (OneShotServiceImpl and PeriodicSyncServiceImpl)
3. New parameter in BackgroundSyncParameter to record the max number of times to wake up Chrome for serving periodic sync registrations.
4. Logic to wake Chrome up periodically, and fire periodic sync events.
5. New WPT, unit and integration tests.
6. Code for new components in third_party/blink
7. periodic_sync.mojom will have different logic to one_shot_sync.mojom:
    a. SyncRegistration will be extended to PeriodicSyncRegistration, which will include an optional SyncOptions, containing a min_Interval to begin with.
    b. DidResolveRegistration will need to include the unique_id.
8. PeriodicSyncManager logic
    a. Check with Site Engagement before scheduling tasks.
    b. Set delay_until appropriately, as discussed in previous sections.
9. Permission logic: Chrome will restrict the feature to PWAs, provided the existing background sync permission is enabled. On Android, we won't check for the existing background sync permission.

Metrics to be collected

See PRD. Detailed list TBD. Some ideas:

1. How often we're waking up Chrome per day, to check if that matches the target of max two times.
2. Whether a registration is periodic or one-shot.
3. How often does site engagement data change, and whether it relevant to us each time.
4. Number of retries made per registration.
5. Number of suspended registrations.

Ideas for v2

1. Updating the service worker before firing an event, for periodic Background Sync only.
2. Skip firing an event when a tab for the origin is open, for both kinds of Background Sync.
3. unregister() method for one-shot Background Sync.
4. Penalize registration whose periodic sync events fail repeatedly.

Privacy and Resource Usage

Please see the PRD for a discussion on these.

References

1. Initial brainstorm doc. (Google internal)
2. PRD (Google internal)
3. One-shot Background Sync design