Events in Workers and Worklets

nzolghadr@chromium.org, majidvp@chromium.org, mustaq@chromium.org
Nov 2018
Status: Version 1.0

Related Chromium bugs:
- https://crbug.com/732881

Problem Definition

Today on the Web Platform only the main thread has access to the DOM elements and receives their related input events. In particular, Web Workers and Worklets do not have access to these. If we allow workers to receive input events for a given DOM node we enable many latency sensitive applications to leverage workers.

As a result if developers want to do some event dependent logic like drawing on an off-screen canvas or using fetch in their worker thread they need to listen to those events on the main thread and postMessage them across to the worker thread. Here the worker is not only going to be blocked by the main thread to handle the events first but also suffers from the latency introduced by the extra thread hop. This is particularly unfortunate for the input events where the latency is important.

Use Cases

1. **Low-Latency Drawing**: Offscreen Canvas now provides workers with an off-main-thread drawing context. However any input driven drawing is still blocked by the main thread getting the user events/inputs and forwarding them to the worker to be drawn on the off screen canvas. With this latency paper-like drawing cannot be achieved on the web pages via workers.
2. **Gaming and XR**: Similar to the drawing use case, low latency input handling and drawing is important for these applications and they can benefit from access to both input and canvas in worker threads.
3. **Page Analytics**: Analytic scripts that process events without needing to access DOM. Some analytics scripts want to send user input over the network and only need access to the raw coordinates and state of the pointer including the buttons. In this case accessing DOM or knowing whether the default actions were prevented is not important.
4. **Interactive Animations**: Animation Worklet enables scripted animation off-main-thread. By providing input events to Animation Worklet we enable rich interactive animation driven by the user input to be off-main thread to ensure smoothness and responsiveness. For many such effects passive access to pointer input events is sufficient[1].
5. **Low-Latency Interactive Audio:** Audio Worklet provides a rich off-main-thread audio context which is being used to enable high-fidelity rich audio applications on the web. If audio worklet were able to process user input (including messages from Web MIDI API) directly it may help reduce the latency from user input to audio output which is important for interactive audio applications[2].

This document proposes an approach to event delegation that addresses the use cases without requiring DOM access in workers. The proposed approach assumes the worker only needs to **passively observe** the events (e.g., pointer position and the state of the buttons) which seems to be sufficient for most of these use cases. But we also briefly discuss alternative designs

[1] Some interactive animations e.g., drag-and-drop may need a way to conditionally prevent native events e.g., (dragstart).
[2] Similar to animations, a small amount of latency (<10ms) between the user input to audio stream can be tolerated The common synthesizer has 7ms~10ms latency from user's input to the actual audio output.

where the worker has more controls in particular stopping event propagation, and/or preventing default action.

Background

This work builds on top of and borrows ideas from previous work which include: Handling Input Outside Main Thread, Input Events for Worker Threads, Event Delegation Into Workers, IsolatedWorker.

Terminology

- **Worker:** We use *worker* as a generic term to refer to any off-main thread component including workers, worklets. Note that these can be backed by dedicated threads or a thread pool.
- **Main**: We use *main* to refer to the main document scope which controls DOM.

Event Delegation

Here we propose event delegation scheme that assumes there is no DOM access in workers. The proposal introduces a generic event delegation mechanism that in theory works for all DOM events. However to be practical we intend to first focus on enabling this for input events starting with PointerEvents. This addresses key use cases and gives us a chance to prove and perfect the model while also leaving the door open to enable delegation for more event types that can benefit from this capability[3].

Passive Event Delegation

Allow workers to observe events in parallel to main thread **without** the ability to stop their propagation or to prevent default. In this design the main thread continues to be the **sole owner** in control of the event and event flow which means there is no need to block main thread on the workers when propagating the event. Similarly, in most situations[4] the worker can process and handle events without delay or coordination with main thread.

This is similar to allowing workers to register a passive event handler with the added restriction that the handler also cannot stop propagation.

[3] For example Media and Streaming related events, WebSocket events, WebXR event, and additional input events (mouse, keyboard)
[4] The choice of Fork Point affects this.

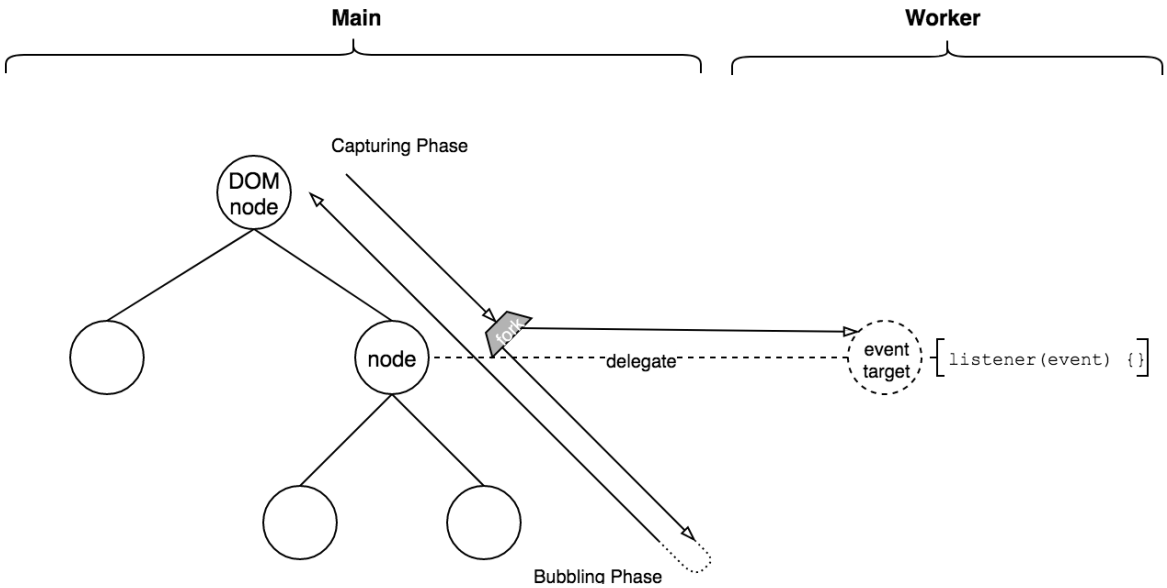

Figure 1. Event flow between main thread and worker: this show fork model (b) but see below for detailed discussion of other fork models.

Considerations on Fork Point

A key consideration in this approach is at which point in the propagation path the event is forked and forwarded to the worker. This fork point choice controls the trade-off between the degree of main/worker control in the delegation process and the performance isolation. Here are some of the most interesting options:

- **a) Fork at root:** Basically fork and delegate the event at the root of propagation path. Here main has no ability to prevent the delegation and the delegated listener always receives the event. It provides maximum performance isolation (no footguns) and as a minor benefit, the delegating agent does not need to compute propagation path before making a decision on whether the event should be delegated. Note that main still controls delegation at target granularity but not at the event granularity. Other solutions in the list provide more granular per event control.
- **b) Fork at target node during capturing phase:** Fork the event only when it would reach[5] the target node. Gives parent nodes and the target node[6] in DOM tree the ability to prevent delegation on the main thread by registering a capturing event listener and preventing propagation. This provides some controls to the main at the cost of making the worker performance only isolated from main when there are no capturing event listener in the propagation path before the target node. In other words, in this model the listeners in the worker are considered capturing listeners that run after main listeners.

[5] It is important to note that this does not necessarily mean that delegation process needs to wait until the event actually reaches the target. For example, if it is clear that there is no possibility of effective preventing propagation (i.e., no capturing event listeners in ancestors on main), the delegation to worker can happen without waiting on event to be dispatched on main.

[6] This assumes delegation happens after all main thread handlers run. We can change this but this seems like a reasonable choice.

- **c) Fork at target node during bubbling Phase:** Similar to model (b) but during bubbling phase. This gives main even more granular control by allowing parent and child nodes the ability to prevent delegations. This loses most of performance isolations since bubbling event listeners are prevalent on the web. In other words, in this model the listeners in the worker are considered bubbling listeners that run after main listeners.
- **d) Worker's choice:** Allow the worker to choose between (b) and (c) based on the type of listener (i.e. capturing vs non-capturing) it registers.

(a) has the strongest performance isolation and perhaps simplest to implement however it provides least amount of control to main and deviates the most from the well-understood DOM Event propagation model. As we move from (a) to (d), we are trading performance isolation in favor of richer control at the main and more alignment with existing event propagation model[7].

We recommend against (c) because it harms the performance isolation which is a key goal of this proposal. Both (a) and (b) are acceptable choices. (d) is interesting since it gives the power to developer but this also means it comes with all the same footguns. Particularly, if the delegetated target is the window node all of these proposals will be equivalent[8].

Proposed API

```
interface mixin EventDelegate {
  void addEventTarget(EventTarget target, EventDelegationOptions? option);
  void removeEventTarget(EventTarget target);
};

// This dictionary allows us to add options in the future if needed.
dictionary EventDelegationOptions {
  any context; // To accommodate for any additional context data from the other thread.
};

interface Worker includes EventDelegate;
interface WorkletAnimation includes EventDelegate;
interface AudioWorkletNode includes EventDelegate;

// The opaque proxy representing an event target in worker.
interface DelegatedEventTarget : EventTarget {};
```

and the API will be used like this:

[7] To implement this in Chromium, we can employ a similar mechanisms for touch rects where we track regions where there are active touch event listeners.
[8] So one idea is to start by only allowing window node to be delegated to the worker and then we have the option to expand to either models later based on the community feedback.

Main thread JS	Worker thread JS

`var t1 = document.getElementById("div1");`
 	
`var myWorker = new Worker("worker.js");`
`myWorker.addEventTarget(t1);`
	`self.addEventListener("eventtargetadded"`
	`,(event) => {`
	`  event.target.addEventListener(`
	`  "pointermove",`
	`  (e) => {`
	`  // Handle event e`
	`  }`
	`});`

The above APIs allow the worker to get the targets separately and add any particular event listener they want to for each of them. This enables the user agent to optimize for the events not required by a worker if they can. Note that the *EventDelegationOptions* is empty for now but it is there to let us add more options to the API in the future.

In the example above when a *pointermove* happens and its target is t1 or any of its **descendants** then user agent also queues the event in the worker queue while still letting the main thread get the event and does all the bubbling and running default actions for such an event. In the model (a) regardless of whether main thread handles the event first and calls stopPropagation or the worker handles it first, they don't interfere with each other's work as in this scenario the worker always listens to the events passively without any side effect on the main thread work and the event is forked at the root.

More detailed example of the API usage can be found in Appendix.

Semantics of *EventDelegate*

An EventDelegate interface is implemented by any component that can acts as a delegate for handling events. Calling *addEventTarget(target)* on a delegate object causes `eventtargetadded` event to be dispatched to the delegate object that has target being the delegated target as argument. The delegate then can register listener against the delegated target (or hold a reference to it to do so at a later time).

Calling *addEventTarget* on the same delegate and with same target is safely ignored avoiding multiple `eventtargetadded` event dispatch.

Calling *removeEventTarget(target)* on a delegate object clear the delegation for that target dispatching `eventtargetremoved` event. This means that any delegated event handler registered on the delegated target will not be called again.

TODO: Discuss lifetime of delegated target. i.e., they are kept alive as long as delegated or if user code holds a reference to them.

Stripping DOM References from Events

As part of this proposal we are providing access to events and event targets in the worker context. To make this work, both the events and targets objects that are delivered to the worker need to have all of their DOM related properties and functions stripped off (or replaced).

Here are some key properties with suggested changes:

Event Target

- dispatchEvent: becomes a no-op for delegated targets in the worker.

Event

- `Target, currentTarget, sourceElement, composedPath`: These are set to null as they might point into other elements that worker doesn't have any reference to. An alternative may be to have a simple opaque ID (See Element.prototype.uniqueID Proposal)
- `stopPropagation(), stopImmediatePropagation(), preventDefault()`: This become a no-op similar to how passive event handler make preventDefault a no-op.

UIEvents

- coordinates: Screen coordinates (screenX/Y) of the event will remain unchanged. Other coordinates may be changed and/or outdated in certain cases, see the section below on computing coordinates.
- `relatedTarget, view`: Set to null.

The list above is focused on PointerEvents properties because as discussed earlier we intend to to start with delegating PointerEvents. However there are a large number of DOM Event types with a varying list of properties that may need to be stripped. One way to scale this is to use a similar approach to Object Serialization for sending objects to a worker. Basically every event type can indicate if it is "delegatable" and specify the process by which an event object can be processed so that it can be used in worker [9].

Note that the above forking scheme can also work for synthetic events but for simplicity sake our initial focus will be on delegating trusted events.

Addressing Double Handling Problem

A general issue with only providing passive event handling is that the event handler on the worker side can not prevent propagation or the default action. This can lead to having multiple

[9] Perhaps "Delegatable" can be the same as "Serializable". Events can be serialized if we allow serialization to drop/replace some of the properties. The advantage is events can also be post messaged which can be useful on its own and makes polyfilling this feature easier.

handlers processing the same event a.k.a double handling when we also have handlers on the main thread.

This is a special case of the more general problem of coordinating actions in the presence of concurrency that exists with any usage of workers. Ultimately dealing with this problem requires application specific logic that uses the underlying platform's concurrency primitives such as message passing. While this proposal makes it easier to have more concurrency in certain area previously not possible but it does not introduce new mechanisms to make dealing with concurrency easier. We think such mechanisms are orthogonal to the proposal here.

A simple and common way to avoid double handling is to have a single handler. Note that this can be achieved fairly without additional syntax:

- Principal handler is main: Developer ensures no action is taken in the worker that interfere with main thread actions.
- Principal handler is worker: Install a handler on main thread that unconditionally prevents propagation or default action or make sure no other handler is run on the main thread.

Note that the latter pattern may benefit from being declarative[10]. This is possible to do by allowing additional options for event delegation or even more broadly for all event handling.

Computing coordinates for dispatched events

Most use cases we mentioned above require event coordinate information, so it's important to be highlight how a Worker can interpret the coordinates. Because the container of the event target is maintained by a different thread possibly running at a different "speed", *in general* it is impossible to guarantee that any coordinate that is relative/connected to the part of the DOM outside the event target would be sensible. Therefore:

- since raw event coordinates are not dependent on the rest of the DOM, dispatched events will have screenX/Y unchanged without any problem,
- but other coordinates are all relative (clientX/Y is relative to viewport, offsetX/Y relative to top-left of event target, pageX/Y relative to top-left of page), so they would be outdated (if they are dispatched unchanged) when the position of event target changes during dispatch (e.g. through scrolling) and/or the involved threads are out-of-sync. See this scenario depicting this general problem.

For Chrome, the good news is that even the relative coordinates can be guaranteed to be up-to-date using the fact that we have a compositor thread is guaranteed to be fast (since it doesn't run any user code). No matter what's the relative speed of the main thread vs the

[10] If browsers know in advance that an event handler always prevents propagation or not then it can optimize the propagation. Note that this is useful for **all** event handlers and not just delegated ones so a generic solution applicate to all event handlers is more desirable here, e.g., a flag in EventOptions dictionary.

event-handler Worker thread, the compositor thread is assumed to hold the “ground truth” of the event target’s position—and this is what the user sees on screen—so we can adjust all relative coordinates accordingly right before dispatching the event to Worker.

The bottom line is that the main thread and and the Worker thread should expect to see different relative coordinates for the same event.

Feature Detection and Polyfilling

Feature detection is fairly straightforward by checking the existence of `AddEventTarget` on the worker object.

To polyfill this feature one can add the missing methods on the worker prototype. At a high level the polyfill will:

- Install appropriate event listeners (capturing or bubbling) at appropriate node (target or document) based on forking model.
- Implement event stripping that takes event and removes/replaces any DOM references.
- Use post message (equivalent) to clone and send the event to worker.

Alternative API Designs

Event Delegation Into Workers documents proposes a similar model but with a different API. The advantage of the API proposed here is that the worker can register individual listeners per each target. Also the set of event types are not predefined and can be declared by workers. Finally Input Events for Worker Threads document contains a comprehensive API design exploration for this model as well.

Implementation Considerations

Here we briefly discuss possible implementation strategies and their implications. *Caveat: This is based on Chromium architecture so it may not be applicable to other engines.*

Basic Implementation

A basic implementation of this proposal can simply perform event forking on main. The implementation involved the following pieces:

- Event Forking Logic: clones and strip event object from DOM references.
- Delegated Event Handler Registry: keeps track of all delegated targets and registered handlers against them.

This is fairly straightforward to implement as it reuses existing hit-testing, event propagation, and worker messaging machinery on the main thread. The delegation can simply be seen as yet

another event handler. However, it does not realize any of the performance isolation benefits of the delegation. This would also be more or less how a javscript polyfill provides this feature.

Optimized Implementation

This implementation aims to perform delegation (for a subset of input events) off-main-thread allowing the worker to receive and handle the event without getting blocked on main thread. This is analogous to how threaded scrolling works in most browsers. In Chromium, the delegation for pointer events can be done in compositor thread. The implementation involved the following pieces:

- Event Forking Logic: The basic implementation version can be used here with little or no change.
- Delegated Event Handler Registry: The basic implementation version can be used here with little or no change.
- Richer Hit-Testing on Compositor Thread: We currently hit-test in simple cases to identify scrollable areas and fallback to main thread in more complex cases. A similar approach may be used here.
- Tracking Areas for Potentially Blocking Main Thread Handlers: Depending on the forking model, we may need to track for a given target if the event needs to be processed by main thread handlers first. This is similar to TouchRect tracking and we may be able to extend the same machinery.

Note that depending on how complete is our hit-testing on compositor, this implementation may not handle all corner cases. A simple approach is to fall back to the basic main thread implementation in those cases. This is similar to our threaded scrolling approach and can ensure we provide good performance for most common use cases without requiring a full re-implementation of hit-testing[11] off main thread.

Active Event Delegation

One can imagine an active delegation model where we expands passive delegation with the ability for worker to **prevent propagation** and **prevent default**. This model enables a richer composition using fairly simple DOM Event propagation model at the cost of sacrificing concurrency as it requires event propagation chain to include both main thread and worker handlers in a single sequence.

We believe this is not the right tradeoff for the majority of current use cases. Perhaps the only exception is in a world where workers have ownership of a DOM subtree and we have fully isolated components. Here one can argue that the additional richness in composability may become more important and more efficient to implement. The DOM in Workers document discusses this in more details.

[11] Other engines may have hit-testing design that makes this easier.

While we don't have concrete details on how this can be achieved but the building blocks needed for passive event delegation (e.g., off-main thread hit-testing and even dispatching) can be used for the richer models.

Appendix

Code Examples

These examples use the APIs proposed in Passive Event Delegation section.

Low Latency Drawing with Off-Screen Canvas

Here is a code to draw user input on a canvas completely off the main thread

index.html (DOM and the main thread)

```
// This canvas could be embedded within an iframe if only the root window events are allowed
to be delegated to the worker.
<canvas id="canvas"></canvas>

<script>
  var worker = new Worker("worker.js");
  var canvas = document.getElementById("canvas")

  var handler = canvas.transferControlToOffscreen();
  worker.postMessage({canvas: handler}, [handler]);
  worker.addEventTarget(canvas);
</script>
```

worker.js (Worker thread)

```
var context;

addEventListener("message", (msg) => {
  if (msg.data.canvas)
    context = msg.data.canvas.getContext("2d");
});

addEventListener("eventtargetadded", ({target}) => {
  target.addEventListener("pointermove", onPointerMove);
});

addEventListener("eventtargetremoved", ({target}) => {
  target.removeEventListener("pointermove", onPointerMove);
});

function onPointerMove(event){
  // Use event.clientX/Y or offsetX/Y to draw things on the context.
  context.beginPath();
  context.arc(event.offsetX, event.offsetY, 5, 0, 2.0* Math.PI, false);
  context.closePath();
  context.fill();
  context.commit();
```

```
}
```

Scale/Rotate Gestures using AnimationWorklet

Here an animation worklet is used to create a component that allows an image to be directly manipulated (e.g., scaled and rotates) by multi-touch gestures.

index.html

```
<img id='target'>

<script>
await CSS.animationWorklet.addModule('worklet.js');
const target = document.getElementById('target');

const rotateEffect = new KeyFrameEffect(
   target, {rotate: ['rotate(0)', 'rotate(360deg)']}, {duration: 100, fill: 'both' }
);
const scaleEffect = new KeyFrameEffect(
  target, {scale: [0, 100]}, {duration: 100, fill: 'both' }
);

// Note the worklet animation has no timeline since the animation is not time-based.
const animation = new WorkletAnimation('scale_and_rotate', [rotateEffect, scaleEffect],
null);
animation.play();

// Delegate pointer events for target to worklet animation.
animation.addEventTarget(target);
</script>
```

worklet.js

```
// Made up library that given pointer event sequence can compute an active gesture similar
to Hammer.js
import {Recognizer} from 'gesture_recognizer.js'


registerAnimator('scale_and_rotate', class {
  constructor() {
    this.gestureRecognizer = new Recognizer();
    this.addEventListener("eventtargetadded", (event) => {
       for (type of ["pointerdown", "pointermove", "pointerup"])  {
          event.target.addEventListener(type, (e) => {
            this.gestureRecognizer.consumeEvent(e)));
          });
       }
   });
  }

  animate(currentTime, effects) {
    // Note that currentTime is undefined and unused.

    // Get current recognized gesture value and update rotation and scale effects
```

```
accordingly.
    const gesture = this.gestureRecognizer.activeGesture;
    if (!gesture) return;

    effect.children[0].localTime = gesture.rotate * 100;
    effect.children[1].localTime = gesture.scale;
  }
});
```