Events in Shadow DOM

2012-07-05
by: Hayato Ito (hayato@chromium.org)

Objective

This document explains how an event dispatch behaves in DOM tree with Shadow DOM. This document is intended to be a supplemental document of the Shadow DOM specification Section 6, 'Events'.

Pre-Requirements

1. Readers should be familiar with the concept of Shadow DOM.
2. Readers should be familiar with 'DOM events'.

If you are not familiar, see the following specifications:

1. W3C Shadow DOM
2. DOM Level 3 Events

Goals

We must achieve the following goals:

1. Attaching a shadow DOM subtree to a node should not affect the behavior of event dispatch in enclosing DOM tree. That means:
    - Users of web components don't have to worry about whether shadow DOM is attached or not. Event listeners on enclosing DOM tree should work in the same way.
    - Authors of web components do not have to worry about changing the behaviour of an event dispatch on enclosing DOM tree.
2. Enforce upper-boundary encapsulation. We must not leak any inaccessible node of shadow DOM subtree through an event object. Event's attributes, such as 'target' (or 'relatedTarget'), should be *re-targeted* to a relative node.
3. We should not bother event listeners on enclosing DOM tree by propagating an *uninteresting* event fired in the shadow DOM subtree.

The biggest challenge is that we should be extremely careful when an event is fired on a node which is distributed into an insertion point. Distributed nodes have the following two aspects:

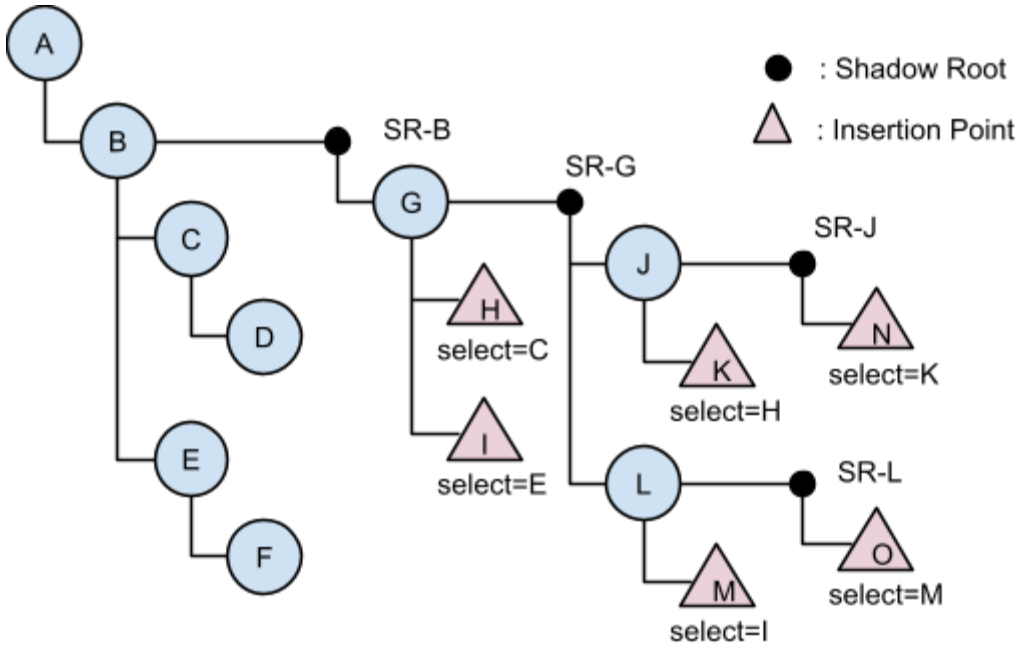
1. From the view of a composed shadow tree, a distributed node is not a direct child of its shadow host anymore. A distributed node behaves as if it was a child of an insertion point. Authors of web components should be able to listen events which is fired on distributed nodes.
2. From the view of an original DOM tree, a distributed node remains a direct child of its

shadow host. The behavior of event dispatch in enclosing DOM tree should not change even if an event is fired on a distributed node.

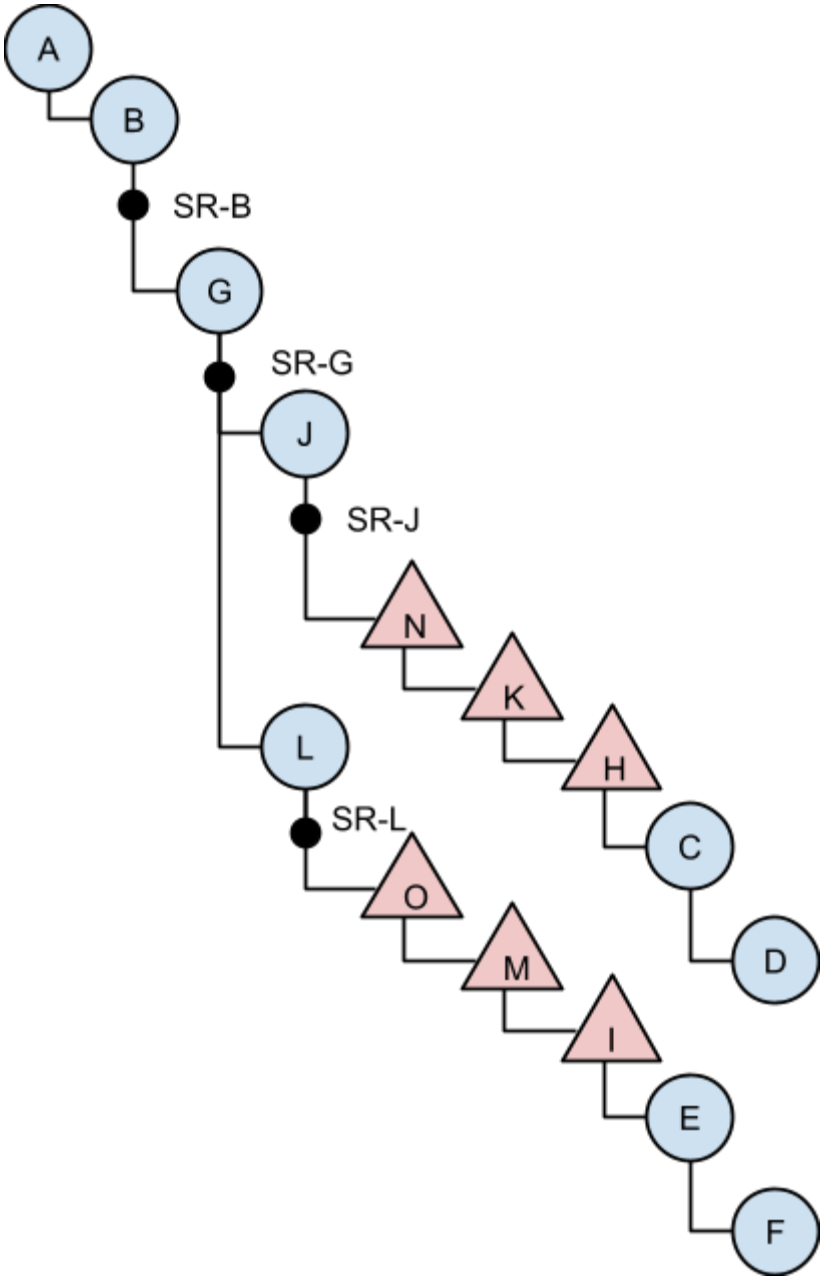
We have to define a behavior which is *the best of both worlds* (a original DOM tree and a composed shadow tree), which would satisfy both web components authors and users at the same time.

Example Tree

Suppose we have the following DOM tree:



Given the DOM tree, a composed shadow tree would be:



Note: This composed shadow tree intentionally includes shadow roots and insertion points so that you can understand how event dispatch behaves easily. In general, a composed shadow tree includes neither insertion points nor shadow roots since these nodes are not used in rendering.

Basic Ideas

- Events should go down (in capturing phase) or up (in bubbling phase) in ancestors chain of a composed shadow tree, instead of an original DOM tree.
- We should retarget target (and relatedTarget if exists) of an event object to a *appropriate* relative node so that we don’t leak any nodes of shadow DOM subtree

to enclosing DOM tree.
Example: Suppose that an event listener is registered on node 'A' and a 'click' event is fired on node 'J'. If event's target is not retargeted, an event listener on node 'A' can see node 'J' by accessing an event.target. That breaks an upper boundary. We should retarget event.target to node 'B', which is a relative target in this case.

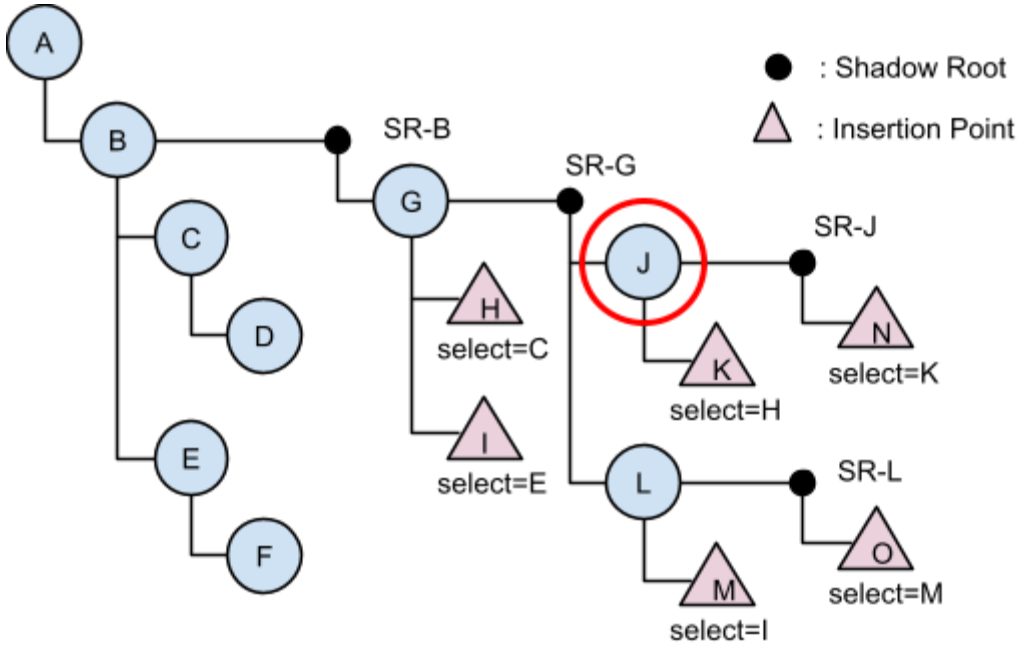
- Not only at an node where an event is originally fired, we should dispatch an event with AT_TARGET phase at some nodes of ancestors if a retargeted target is identical to the node.
- We should not dispatch some kinds of events on a shadow host (and its ancestors) if both an event's original target and an event's original relatedTarget are *within* the shadow host.
  Example: Suppose a mouse moves from a node to other node within a shadow host. An inner shadow DOM subtree might be interested in such a mouse movement, but the shadow host should not be bothered by such a mouse movement.

Examples:

Let's see examples before introducing algorithms because most readers might not be interested in concrete algorithms and want to know how event dispatch behaves quickly.

Case1: Node J is clicked.

A 'click' event will be fired (target: J).

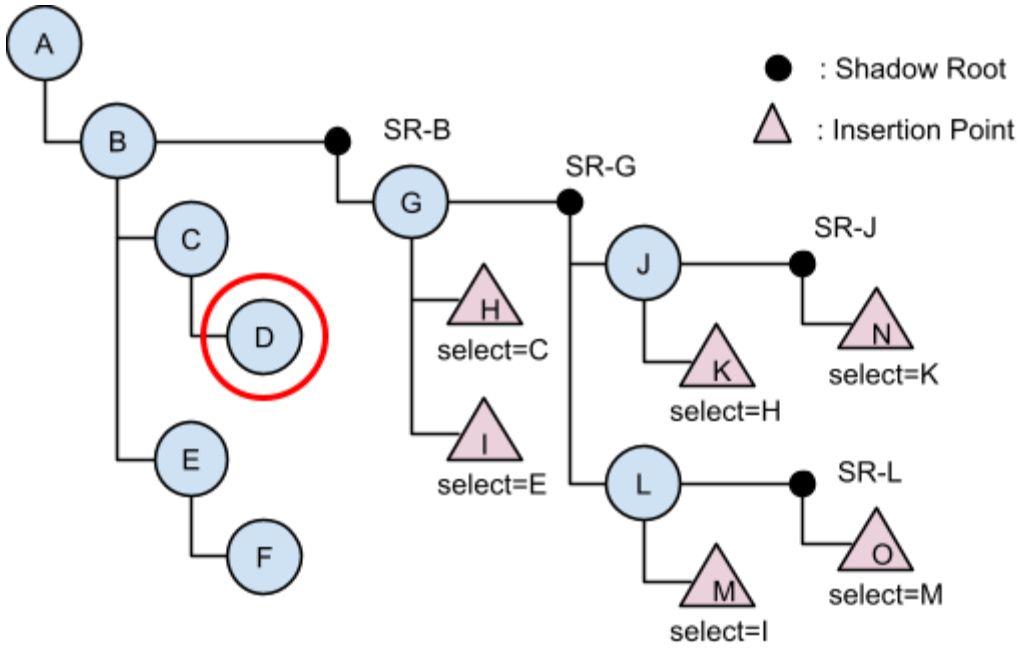


Ancestors of composed shadow tree should receive a 'click' event, but event's target should be retargeted at some nodes as follows:

currentTarget	target
J	J
SR-G	J
G	G
SR-B	G
B	B
A	B

Case2: Node D is clicked.

A ‘click’ event is fired (target: D).



A node D is distributed. Every event ancestors of node D in composed shadow tree can receives an event, including insertion points.

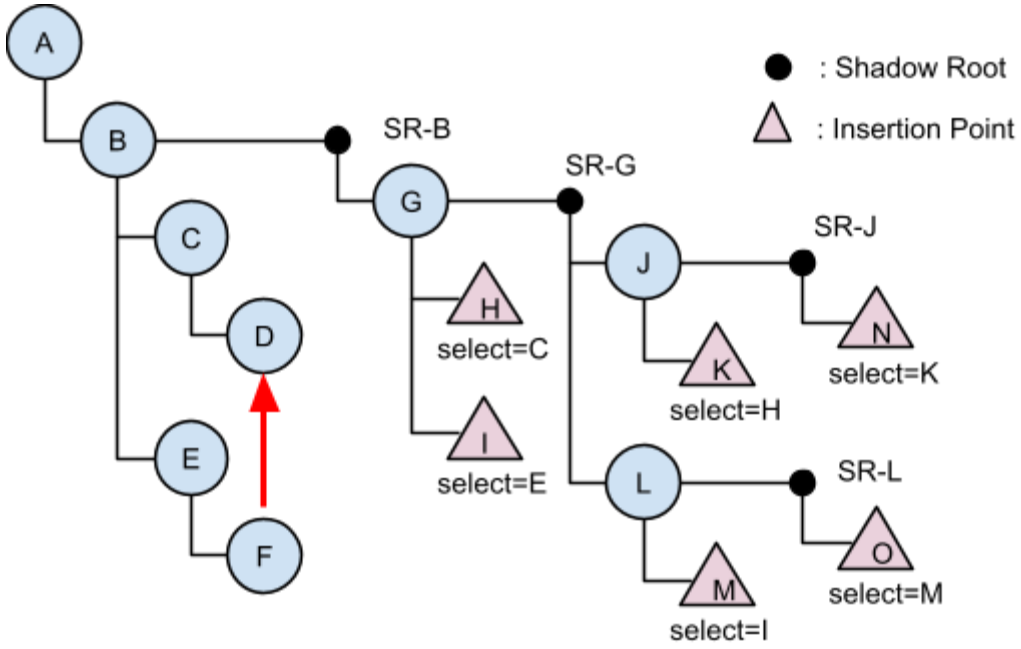
currentTarget	target (after retargetting)
D	D
C	D
[H]	D

[K]	D
[N]	D
SR-J	D
J	D
SR-G	D
G	D
SR-B	D
B	D
A	D

Since node ‘D’ is accessible from every nodes in this case, retargeting does not happen.

Case3: Mouse moves to node D from node F.

A ‘mouseover’ event is fired (target: D, relatedTarget: F).



Both a target node and a relatedTarget node are distributed.

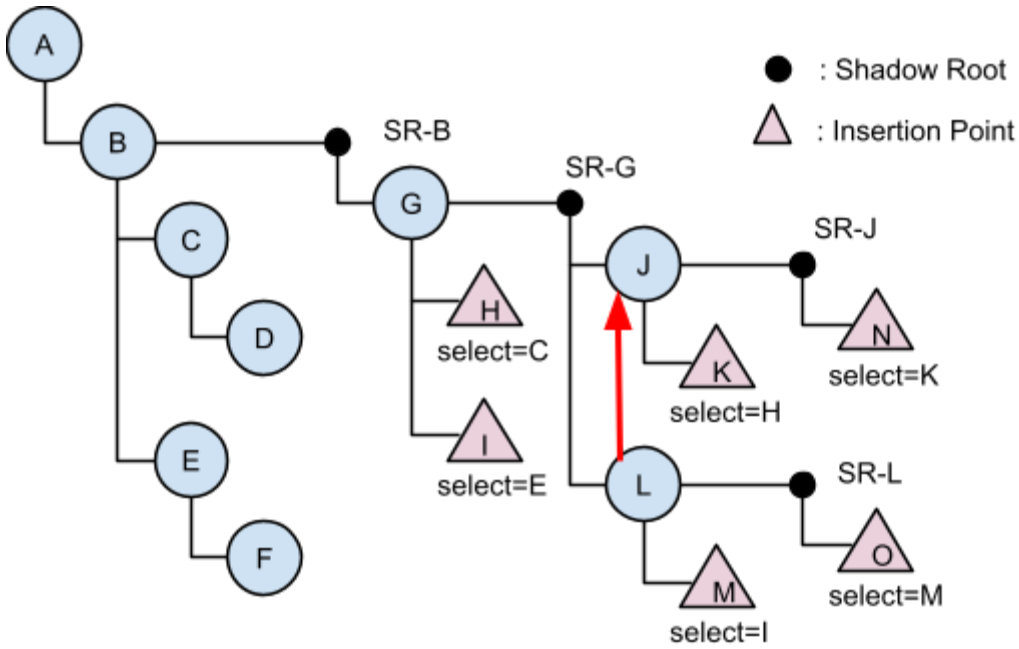
currentTarget	target	relatedTarget
D	D	F

C	D	F
[H]	D	F
[K]	D	F
[N]	D	F
SR-J	D	F
J	D	F
SR-G	D	F
G	D	F
SR-B	D	F
B	D	F
A	D	F

Every nodes can receive an event with target=D and relatedTarget=F. These nodes are accessible from every nodes. We don't have to retarget.

Case4: Mouse moves to node J from node L.

A 'mouseover' event is fired (target: J, relatedTarget: L).



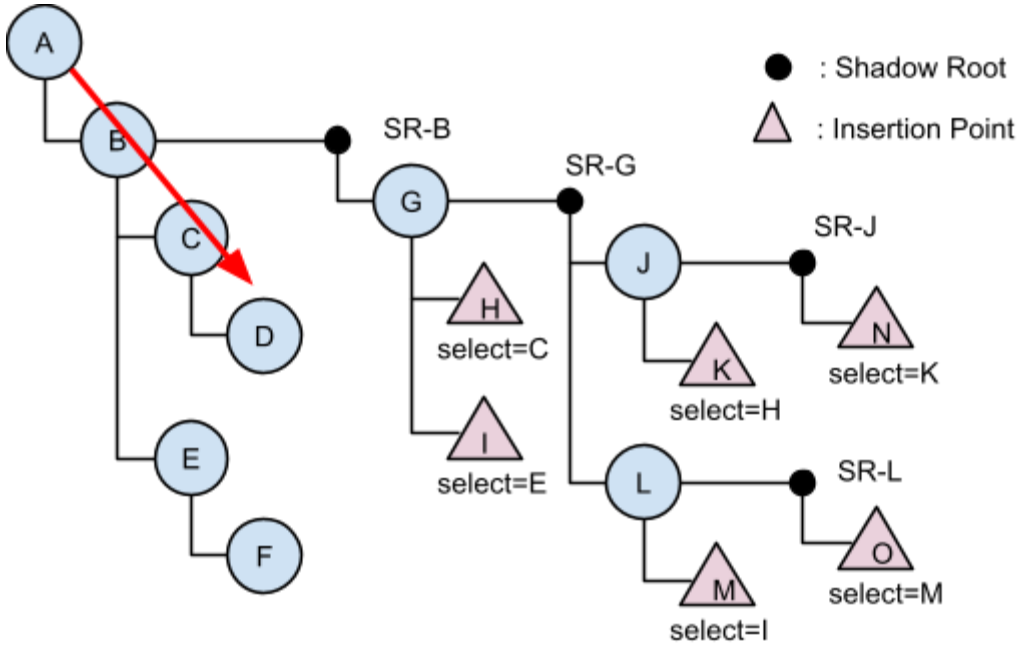
In this case, an event should not propagate to outside of shadow DOM subtree. The shadow host ‘G’ is not interested in movement of mouse within itself.

currentTarget	target	relatedTarget
J	J	L
SR-G	J	L

That’s all. A node ‘G’ (and it’s ancestors) does not receive a ‘mouseover’ event.

Case 5: Mouse moves to node D from node A.

A ‘mouseover’ event is fired (target: D, relatedTarget: A).



A mouse moves within the original DOM tree. But target node, ‘D’, is distributed.

currentTarget	target	relatedTarget
D	D	A
C	D	A
[H]	D	A
[K]	D	A
[N]	D	A
SR-J	D	A

J	D	A
SR-G	D	A
G	D	A
SR-B	D	A
B	D	A
A	D	A

Case 6: Mouse moves to node A from node D.

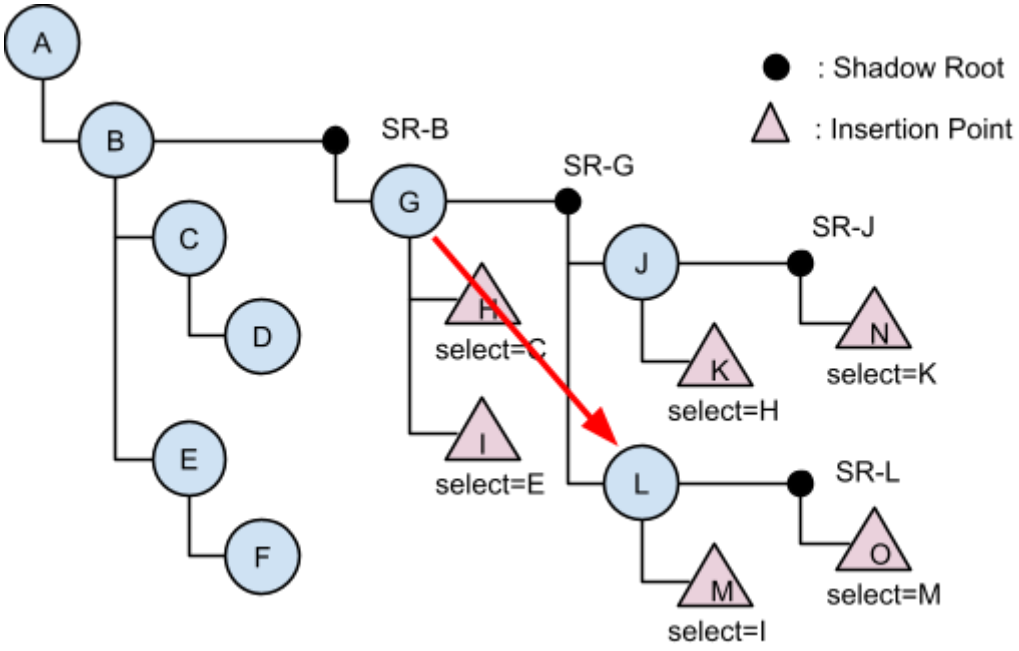
A ‘mouseover’ event is fired (target: A, relatedTarget: D).

This is the opposite case of case5.

currentTarget	target	relatedTarget
A	A	D

Case 7: Mouse moves to node L from node G.

A ‘mouseover’ event is fired (target: L, relatedTarget: G).



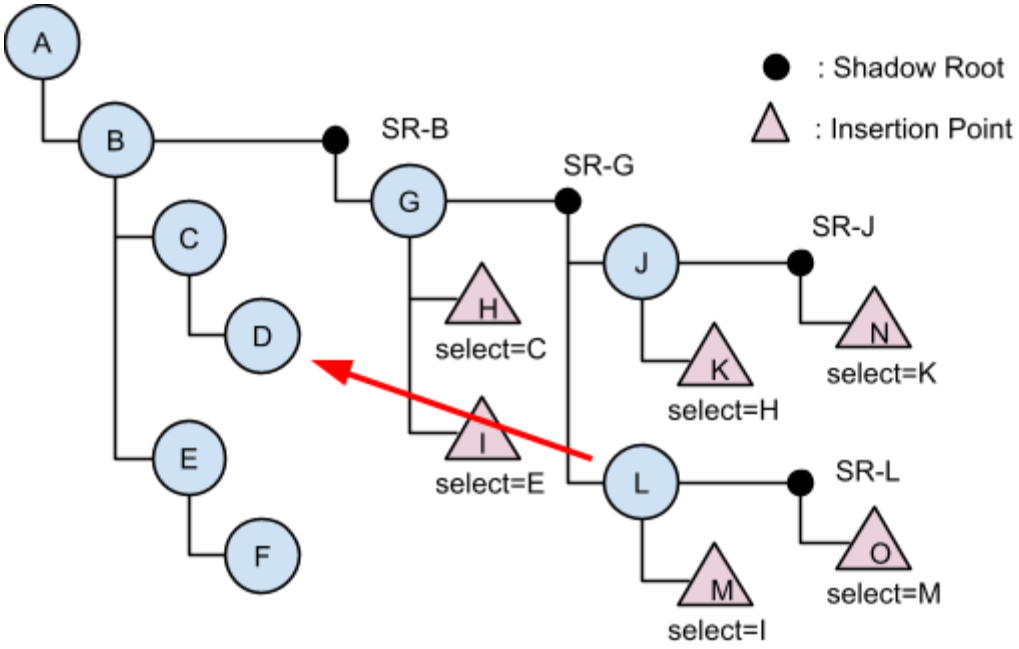
A mouse moves to a node of shadow DOM subtree from its host.

currentTarget	target	relatedTarget
L	L	G
SR-G	L	G

That's all. A shadow host, 'G', should not receive an event.

Case8: Mouse moves to node D from node L.

A 'mouseover' is fired (target: D, relatedTarget: L).



This is a tricky case because some nodes must not see an original relatedTarget (node 'L'). So we have to retarget relatedTarget as follows:

currentTarget	target	relatedTarget
D	D	B
C	D	B
[H]	D	G
[K]	D	L
[N]	D	L
SR-J	D	L
J	D	L
SR-G	D	L
G	D	G
SR-B	D	G
B	D	B
A	D	B

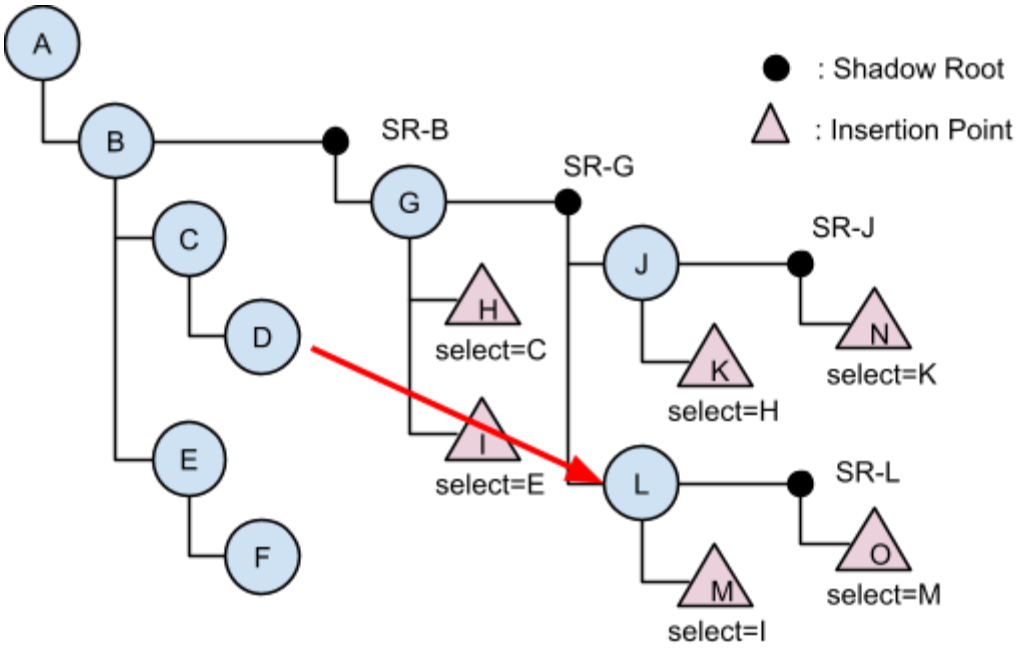
Read carefully how relatedTarget is retargeted on each nodes. Later, I'll introduce an algorithm which explains how these retargeting can be achieved.

From the view of enclosing DOM tree, it's worth noting that the event dispatch behaves as if there was no shadow DOM attached. The event dispatch behaves as if a mouse moved to node 'D' from node B' as follows:

currentTarget	target	relatedTarget
D	D	B
C	D	B
B	D	B
A	D	B

Case9: Mouse moves to node L from node D.

A 'mouseover' event is fired (target: L, relatedTarget: D).



This is the opposite case of case 8.

currentTarget	target	relatedTarget
L	L	D
SR-G	L	D
G	G	D
SR-B	G	D

B	B	D
A	B	D

From the view of enclosing DOM tree, the event dispatch behaves as if a mouse moved to node ‘B’ from node ‘D’.

Note that there is a symmetry relationship between case 8 and 9. Let’s compare nodes which appears on both cases.
Case 8:

currentTarget	target	relatedTarget
SR-G	D	L
G	D	G
SR-B	D	G
B	D	B
A	D	B

Case 9:

currentTarget	target	relatedTarget
SR-G	L	D
G	G	D
SR-B	G	D
B	B	D
A	B	D

On each nodes, target and relatedTarget are just swapped on case 8 and case 9.

Algorithms

The section 6, ‘Events’, of Shadow DOM specification explains concrete algorithms. Let me quote some important algorithms here:

Event Retargeting algorithm

The retargeting algorithm is used to determine relative targets, and it must be equivalent to processing the following steps:

Input

NODE, a DOM node

Output

TARGETS, a list of tuples, each containing NODE's ancestor and its relative target

1. Let STACK be a stack of DOM nodes
2. Let ANCESTOR be NODE
3. Let LAST be **undefined**
4. Repeat while ANCESTOR exists:
    a. If STACK is empty, push ANCESTOR into STACK
    b. Otherwise, if ANCESTOR is an insertion point:
        i. If LAST is distributed or assigned into ANCESTOR:
            1. Let TARGET be the DOM node at the top of STACK
            2. Push TARGET into STACK
    c. Let TARGET be the DOM node at the top of STACK
    d. Add (TARGET, ANCESTOR) tuple to TARGETS
    e. If ANCESTOR is a shadow root, pop STACK
    f. Let LAST be ANCESTOR
    g. Set ANCESTOR to be the result of parent calculation algorithm, given ANCESTOR as input

Retarageting relatedTarget algorithm

The related target resolution algorithm **must** be used to determine the value of the relatedTarget property and **must** be equivalent to processing the following steps:

Input

NODE, the DOM node on which event listeners would be invoked

RELATED, the related target for the event

Output

ADJUSTED, the adjusted related target for NODE

1. Let TARGET be NODE
2. Let ADJSUTED be **undefined**
3. Repeat while TARGET exists:
    a. Let STACK be an empty stack of DOM nodes
    b. Let ANCESTOR be RELATED
    c. Let LAST be **undefined**
    d. Repeat while ANCESTOR exists:

        i. IF STACK is empty, push ANCESTOR into STACK
        ii. Otherwise, if ANCESTOR is an insertion point:
            1. If LAST is distributed or assigned into ANCESTOR:
                a. Let HEAD be the DOM node at the top of the STACK
                b. Push HEAD into STACK
        iii. If ANCESTOR and TARGET are in the same subtree:
            1. Let ADJUSTED be the DOM node at the top of the stack
            2. **Stop.**
        iv. If ANCESTOR is a shadow root, pop STACK
        v. Set ANCESTOR to be the result of parent calculation algorithm, given ANCESTOR as input
    e. If TARGET is a shadow root, let TARGET be the shadow host of TARGET
    f. Otherwise, let TARGET be TARGET's parent node

We use a 'stack' here to keep track of 'the best target node' in each scope. I won't explain how these algorithms work well actually here. I recommend you to simulate algorithms step by step using some examples.
Although I've not explained the case of hosting multiple shadow roots to avoid making the examples more complex, these algorithms are carefully designed to work well for multiple shadow roots.

Browser vendors don't have to implement this algorithm as is. Feel free to optimise the performance as long as an implementation can achieve an equivalent result of the algorithms.

I've used some performance optimization technique when I implemented these algorithms in WebKit so that we can avoid unnecessary computational complexity. If you are interested in, See EventDispacher.cpp in WebKit.

If you find any suggestions, feel free to file a bug to a Shadow DOM spec, section 6.

Appendix:

Specifications:

1. Shadow DOM: https://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/shadow/index.html
2. DOM Level 3 Events: http://www.w3.org/TR/DOM-Level-3-Events/

Related bugs on W3C Shadow DOM specification:

- Bug 16176 - [Shadow]: What should we do if an event happens on light child which is distributed to a insertion point.
- Bug 16599 - [Shadow]: Event Dispatch on non-distributed light children.
- Bug 17090 - [Shadow]: Listening to specific nodes, distributed to insertion points is hard.

Related bugs on WebKit:

- Bug 78586 - Event dispatching should use the composed shadow DOM tree instead of original DOM tree.
- Bug 89073 - Modify event re-targeting algorithm so that we can tell which distributed node is clicked.