Implement BiDi sessions in ChromeDriver

This Document is Public

Attention: Externally visible, non-confidential
Author: nechaev@chromium.org
Status: Inception | Draft | Accepted | Done
Short Link: go/chromedriver-bidi-sessions
Created: 2022-01-25 / **Last Updated**: 2022-04-05

One-page overview

Summary

We propose implementing a BiDi session as an extended Classic session that delegates BiDi commands to the BiDi Mapper. The client communication, both HTTP and WebSocket, will be performed by the IO thread. The communication between ChromeDriver and Chrome will be mediated by the BiDi Mapper.

Platforms

All platforms officially supported by ChromeDriver.

Team

nechaev@chromium.org, sadym@chromium.org, mathias@chromium.org

Tracking issue

Tracking issue for milestone #1 (2022Q1) is available in chromedriver:4016.

Value proposition

BiDi sessions support will enhance user experience in e2e testing and facilitate further development of WebDriver BiDi standard.

Code affected

ChromeDriver

Signed off by

Name	Write (not) LGTM in this row
mathias@chomium.org	
sadym@chromium.org	
johnchen@chromium.org	

Core user stories

ChromeDriver is a server that has only one type of client - an application sending commands to it. The commands can arrive over an HTTP channel (WebDriver Classic) and over a WebSocket channel (WebDriver BiDi). This document concerns BiDi session implementation.

We will use the following words interchangeably:

- User and Client denote the client application.
- Server, WebDriver and ChromeDriver stand for the ChromeDriver instance serving the client commands.
- Browser and Chrome denote the Chrome / Chromium browser
- Mapper denotes the BiDi Mapper - the BiDi implementation loaded by ChromeDriver to the Chrome process.

There are two types of connections:

- The connection between the user app and ChromeDriver. We will call it user connection or BiDi connection.
- The connection between ChromeDriver and the browser. We will call it a browser connection or CDP connection.

The BiDi standard provides two distinct types of BiDi sessions. We will name them as follows:

- **HTTP-BiDi** session is created and destroyed over an HTTP command. This session accepts both HTTP and BiDi commands.
- **Pure-BiDi** session is created via 'session.new' WebSocket command. This session does not support any HTTP commands. The latest requirement (no HTTP support) does not seem to be mandatory - we can skip it if this can simplify our implementation.

User Story 1: Create a connection and a session simultaneously (HTTP-BiDi session)

User sends an HTTP POST request with uri '/session' similarly to the classic new session command. The only difference is that the request has the capability webSocketUrl=true.

The server creates a new session, constructs a listener-url of form ws[s]://<host>:<bidi-port>/session/<session_id> and starts / continues listening on the end point <ip>/<bidi-port>.

The server returns a response with the value of listener-url set to webSocketUrl capability. The response also contains sessionId of the new session as specified here.

The user connects to the bidi service via the url in webSocketUrl capability of the response.

The resulting connection is bound to the session with id=<session_id>. This means that any following session command going through this connection will be attributed to the session with id=<session_id>

User Story 2: Create a connection not bound to any session (Pure-BiDi session workflow)

User sends connection request over url of shape ws[s]://<ip>:<port>/session/.

The server creates a new connection not bound to any session and reads for the incoming messages. Only commands without session context (session.new, session.status and alike) are allowed. Any commands requiring session context are treated by the server as erroneous.

User Story 3: Create a session from a connection not bound to any session (Pure-BiDi session)

Precondition: User has already executed User Story 2 and has an unbound (to any session) connection.

User issues the command 'session.new' over this connection.

A new session is created by the server and its id is returned to the user in the response.

Now the connection is bound to a session and the user may send any commands requiring session context.

User Story 4: Create a connection attached to an existing session (both kinds of sessions)

Precondition: User has executed either User Story 1 or User Story 3. Session with <session_id> exists on the server side and the user knows <session_id>.

The user can create a new connection via the url ws[s]://<ip>:<bidi-port>/session/<session_id>. This connection will be bound to the existing session with id=<session_id>.

User Story 5. Close a session (HTTP-BiDi session)

The user can send the classic DELETE /session/<session_id> command to close any session.

The session is closed which means that

- All connections belonging to that session are shutdown

- Chrome instance is closed
- All session related information on the server side is deleted.

As standard does not yet specify a BiDi command for closing a session it seems reasonable to allow a session created within the Pure-BiDi workflow to be closed by an HTTP command in the same way as any session. There is an open discussion on this topic.

User Story 6. Send an HTTP command (HTTP-BiDi session)

The user can send any WebDriver Classic command to the HTTP-BiDi session over HTTP. It must be handled with the same result as if it was sent to a classic session.

User Story 7. Send a WebSocket command (both kinds of session)

The User can send any BiDi command to any BiDi session. The command is handled asynchronously - ChromeDriver must accept a new BiDi command even if the last BiDi command is still in progress.

BiDi command handling by ChromeDriver while an HTTP command is not yet finished is NOT required / promised by this document.

User Story 8. Propagate a BiDi event (both kinds of session)

BiDi events arriving over CDP must be propagated over the connection that subscribed to this specific type of events.

Assumptions

- BiDi session must handle both HTTP and BiDi commands (inferred from earlier design docs)
- BiDi command does not have to be accepted before the last HTTP command is handled by the session.

Design

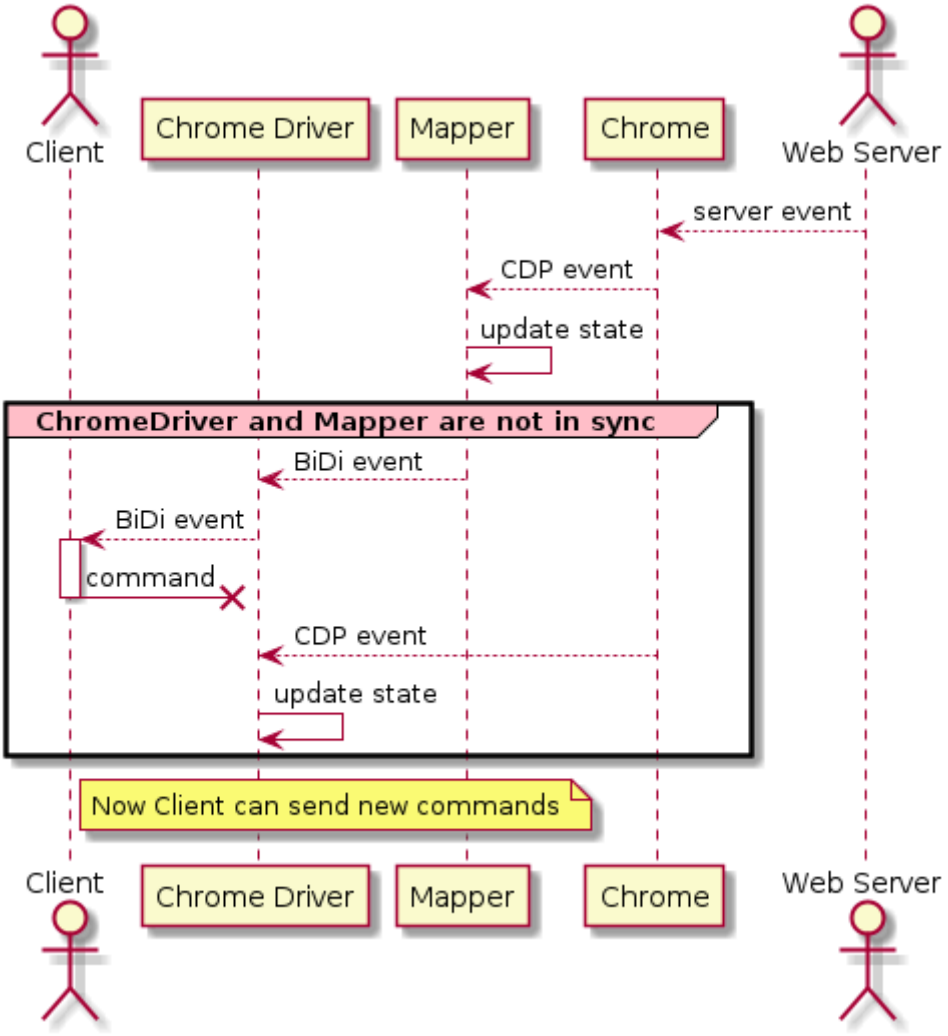
We have agreed to implement BiDi via a Mapper deployed by ChromeDriver into Chrome. If this solution does not work we will explore others.

The most challenging requirement to handle is that a BiDi session created over HTTP must handle both HTTP and WebSocket requests. This in turn requires that the ChromeDriver session state never lags behind the state observable by the client via the events.

Mapper per connection vs. Mapper per session

Single mapper per connection makes message forwarding between User and Chrome pretty straightforward: we store a table of correspondence and route messages in accordance with it.

The possible disadvantage of this approach is that some event coming from the browser might reach the user before the classic session state is updated. In this case user response to the event via an HTTP command will not work as expected. See the illustration below



Another issue is that sequences of events from different mappers do not have to have a certain order between the sequences.

For example we have two mappers M1 and M2. They transmit the following subsequences of events originated from some common sequence:
M1: e1, e2, e3, e4, e5
M2: e1, e3, e4
The events can arrive from Mapper to ChromeDriver in this order: e1, e2, … e5 (all from M1), e1, e3, e4 (all from M2).
This can happen, for instance, if the thread where M2 lives has been preempted.
All these events can arrive before or after the ChromeDriver session state has been updated to reflect the browser state after the event e5. The session state can also be updated up to e5 somewhere in between.

This illustrates that the state that the client can infer via different channels can significantly differ from the ChromeDriver session state.

However, if we ensure that any event is handled by the ChromeDriver session first and forwarded to the client only after this, then the session state will never lag behind the state inferred by the client. This is acceptable because the client assumes that the session might advance before they receive and handle the events.

As the session must not forward the events, blindly acting as a router, multiple CDP connections or multiple Mappers give us no advantages.
Therefore it is suggested to have one CDP connection and one Mapper per session.

The main disadvantage of this architecture is that ChromeDriver must keep the information about each connection and its events of interest to forward only the appropriate events. There is also a need to remember what commands have come over which connection and route the responses accordingly.

Session state is never behind the user

In order to ensure that the session state never lags behind the state observable by the user we must route all the CDP events via the Mapper.

For each CDP event the Mapper does the following:

- Forward the CDP event to ChromeDriver. At this point ChromeDriver will update the session state.
- Update the internal state of the Mapper.
- Generate BiDi events and send them to ChromeDriver. These events will be bluntly retransmitted to the user.

Thus we always have a guarantee that HTTP session state is updated before the BiDi events are propagated to the client.

Mapping between WebSocket connections and threads

Interaction with DOM from multiple threads can create races and therefore it is not supported by CDP. The user should view multiple connections as being bound to a single thread shared among them. In particular this means that a blocking call made via one connection prevents any progress for the calls made via another connection while the first call is being processed. If code executed via one connection actively awaits for a condition set by the code executed via another connection (maybe even from another thread) it might never exit. This kind of problem is considered as a bug in the user code.

Fallback on Classic command

It might happen that some BiDi commands are too difficult or too error prone to be implemented in the Mapper. This, for instance, can be the case with the commands relying on special labels left in DOM elements by ChromeDriver.

All the proposed designs allow falling back on the existing Classic commands.

Proactive event consumption

Current implementation consumes browser events lazily: the session sleeps until the user issues a new HTTP command. This is not acceptable for BiDi as the events must propagate to the user proactively.

The following designs address this issue. As a side effect the Classic session will also consume events instantly. This must improve the Classic session performance.

Preferred implementation

We only consider the possible implementations that can be constructed from the existing building blocks. This is not a big limitation because they already have all the necessary properties, in particular they support non-blocking IO.

We have three types of message pump:

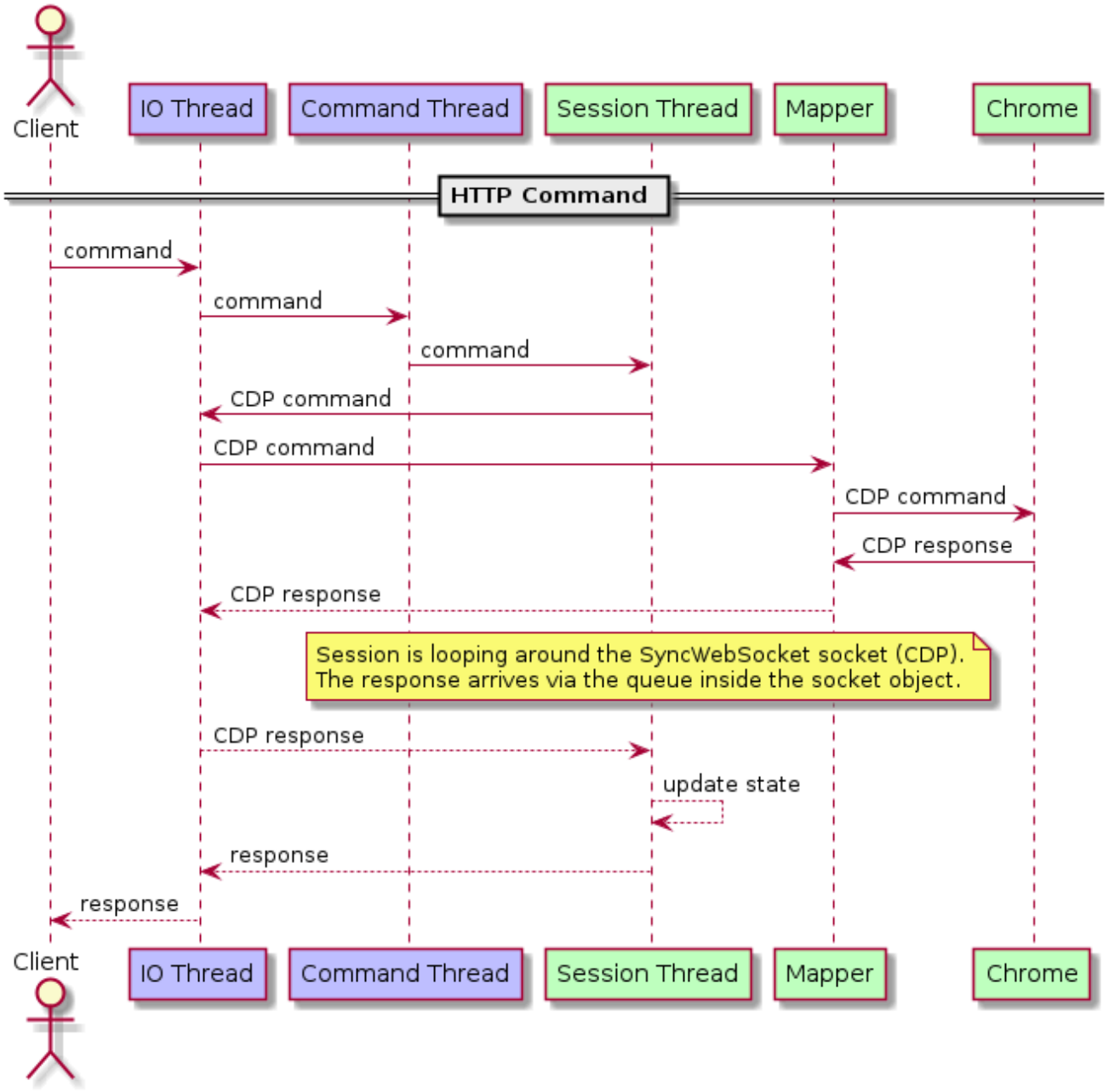
- GUI - supports non-blocking IO and system events
- IO - supports non-blocking IO
- Default - supports commands and timers.

The current implementation of ChromeDriver already uses the IO message pump for the IO thread and the Default message pump for the session threads. The non-blocking IO (sockets, servers) relies heavily on the underlying IO message pump. The blocking IO like SyncWebSocket is built on top of the non-blocking IO: the data is posted to the IO thread and the calling thread awaits for the signal supplied together with the data. The designs below rely only on the existing IO facilities.

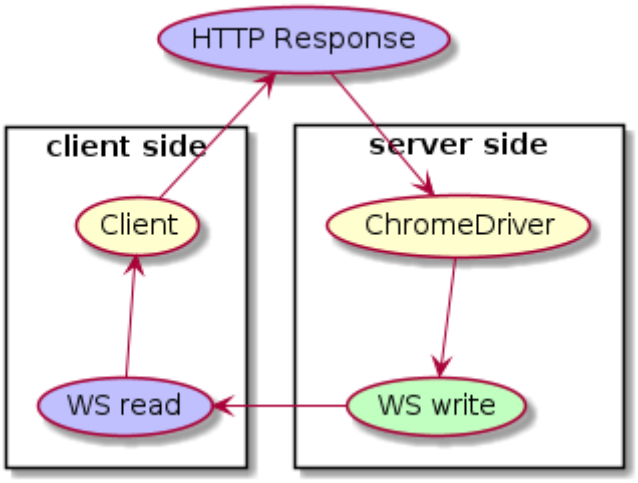
Earlier we considered several designs not-based on the aforementioned building blocks. These have been discarded and will not be mentioned in the final version of the design doc.

We also do not consider any designs where communication between ChromeDriver and Browser significantly differs from the existing implementation as the effort for changing it would be excessive. The only difference that we introduce is that all communication between ChromeDriver and Chrome is mediated by the Mapper.

HTTP command execution will be the same for all designs.



Communication between ChromeDriver and User must be non-blocking in order to avoid deadlocks that can happen when one side tries to write to the socket while the incoming buffer on the other side is full. This is the illustration of a possible deadlock situation.



The proposed design is described as Alternative 3 “Client communication over IO thread”. It is chosen among the others because it provides the most straightforward way to implement a Pure-BiDi session.

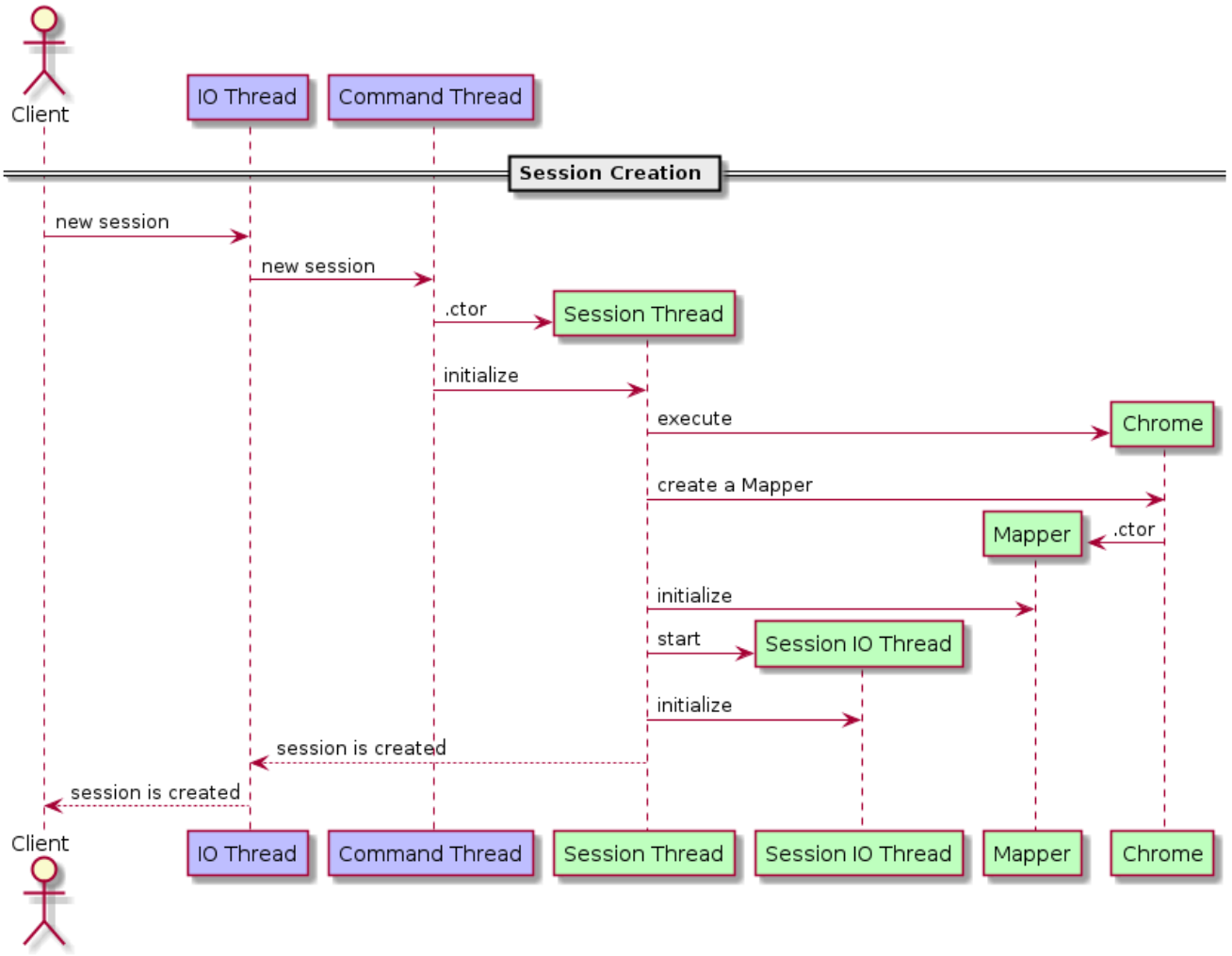
Alternatives considered

The illustrations below have the following color coding:

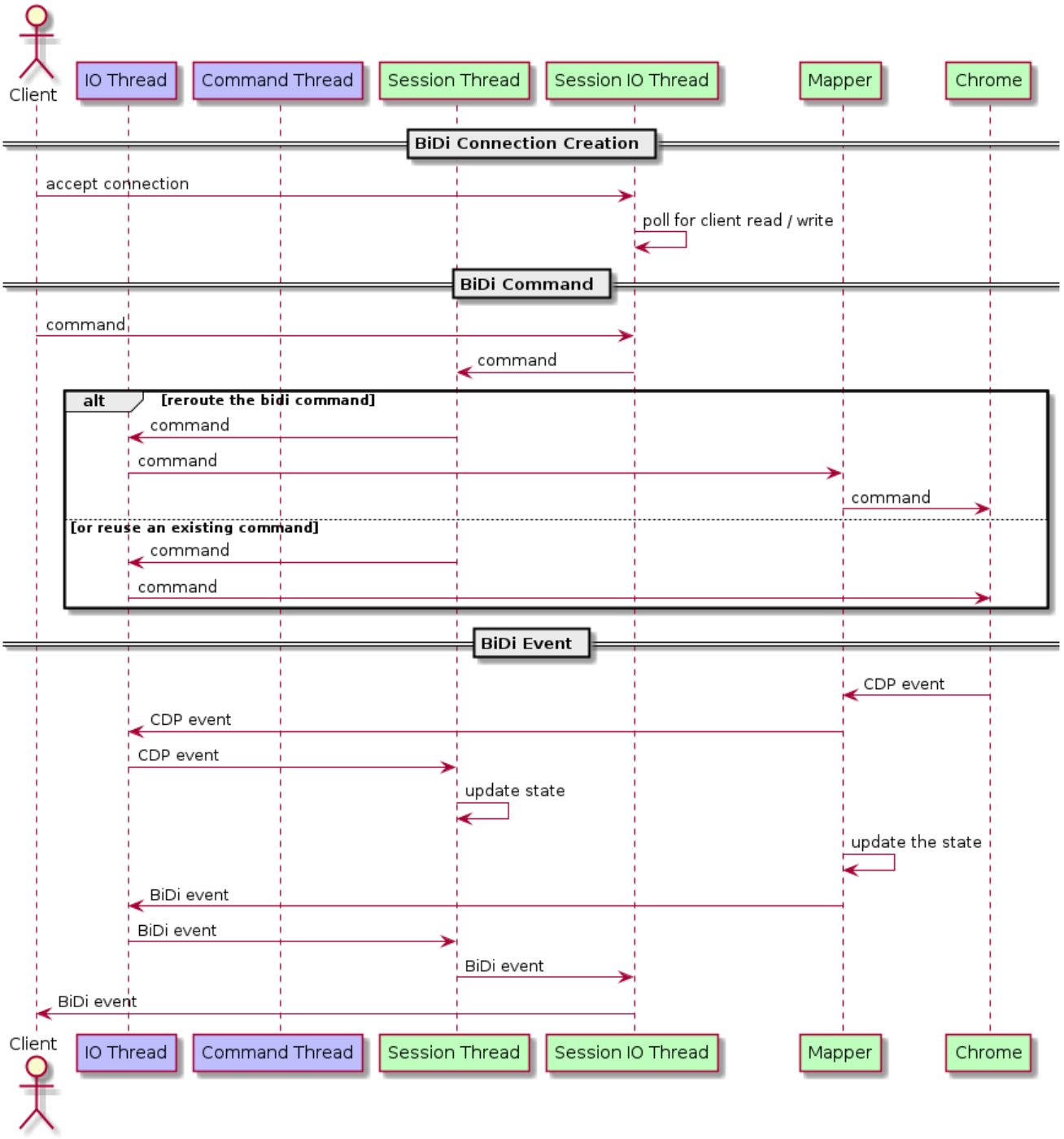
- Violet - threads with lifetime bound to the lifetime of ChromeDriver process
- Green - threads with session lifetime

Alternative 1. Client communications in SessionIO Thread

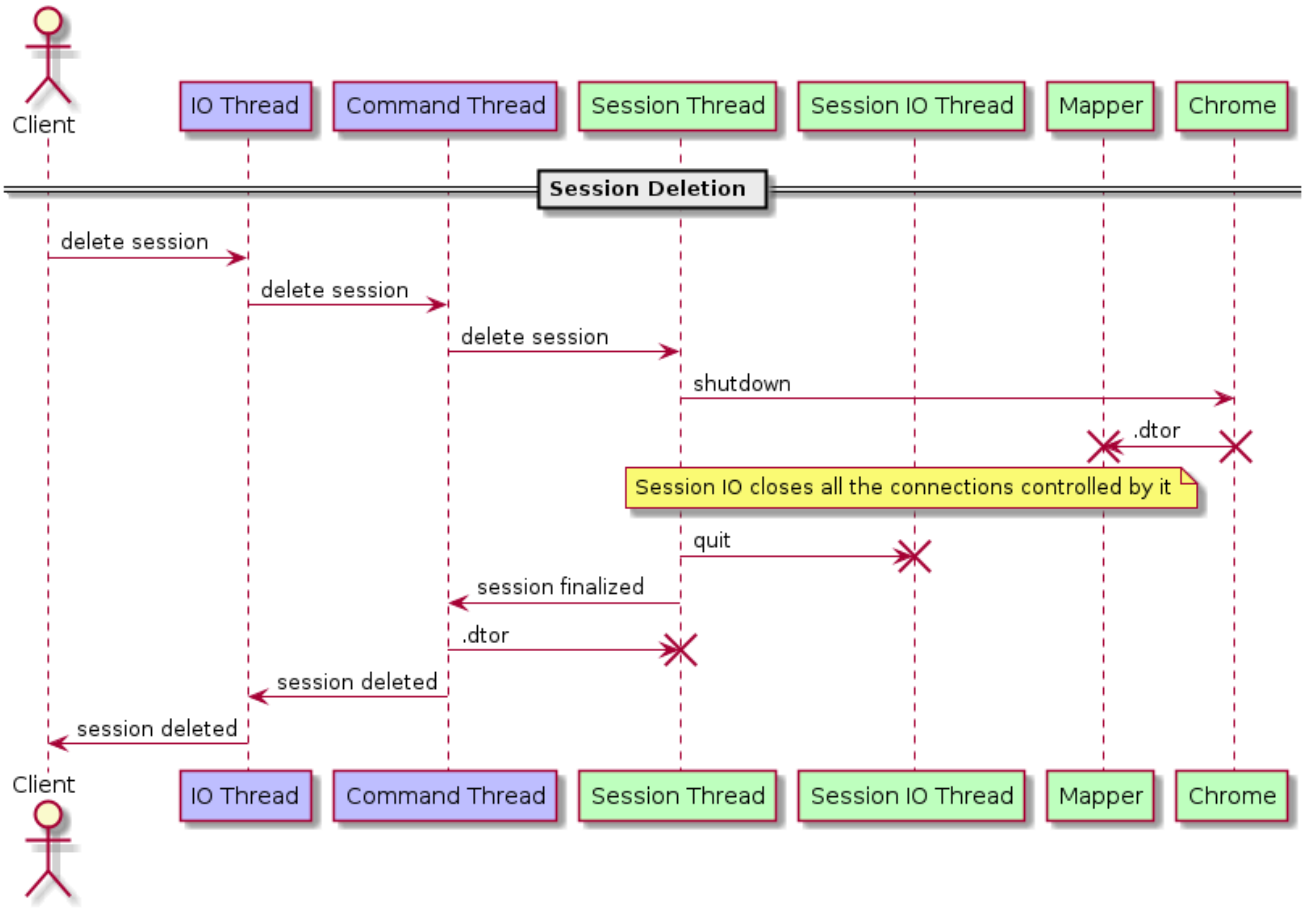
When we create a session we create a thread that will serve the user connections belonging to that session.



The user stories are handled as follows:



The lifetime of the Session IO Thread is bound by the lifetime of the Session Thread.

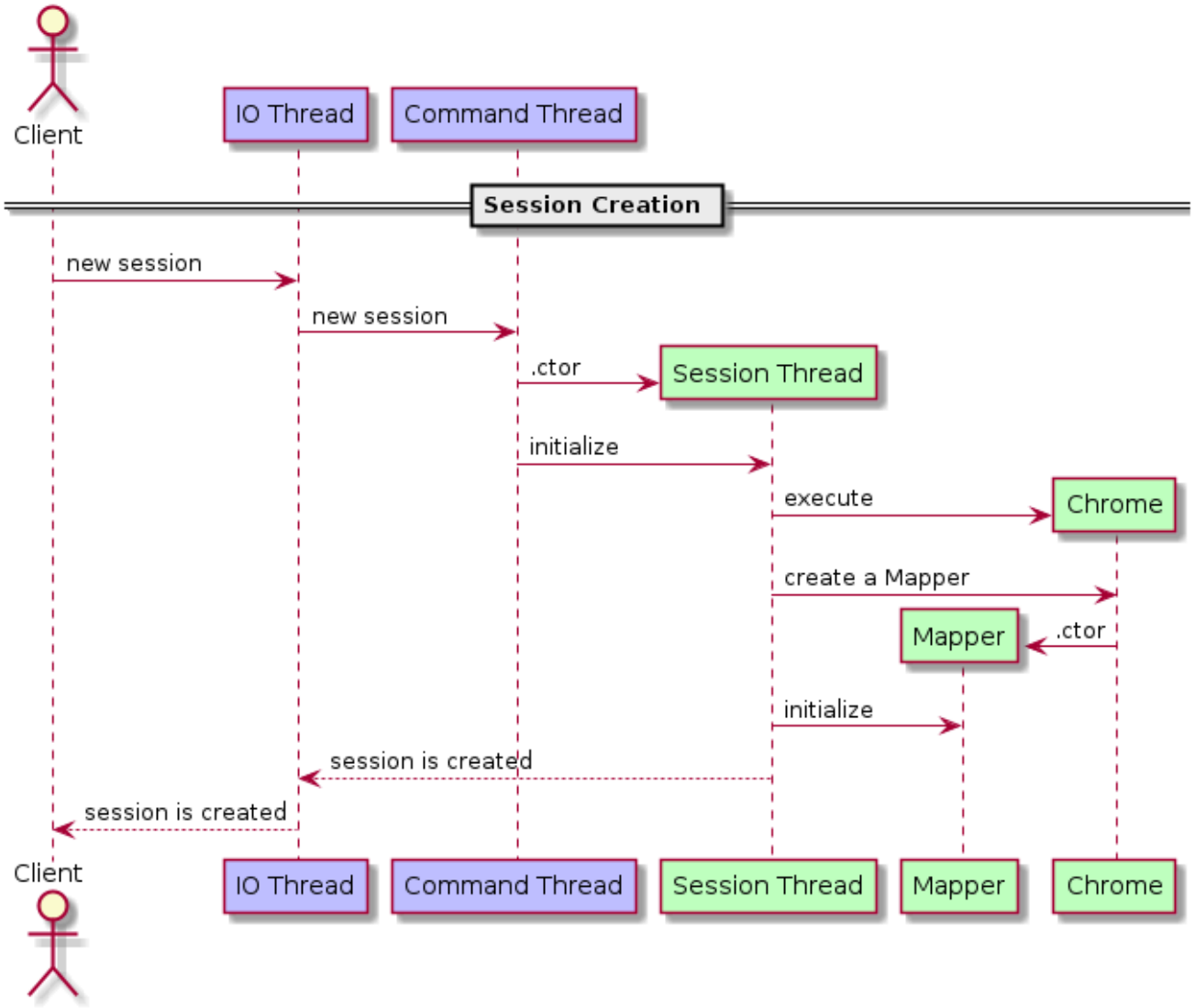


The unbound connections from Pure-BiDi apparently cannot be served by the SessionIO Thread. These, in principle, might be accepted by the IO thread or by some dedicated thread. An unbound connection has to migrate to the SessionIO Thread as soon as a new session is created via this connection. Such connection migration is not supported by the current implementation of HttpServer.

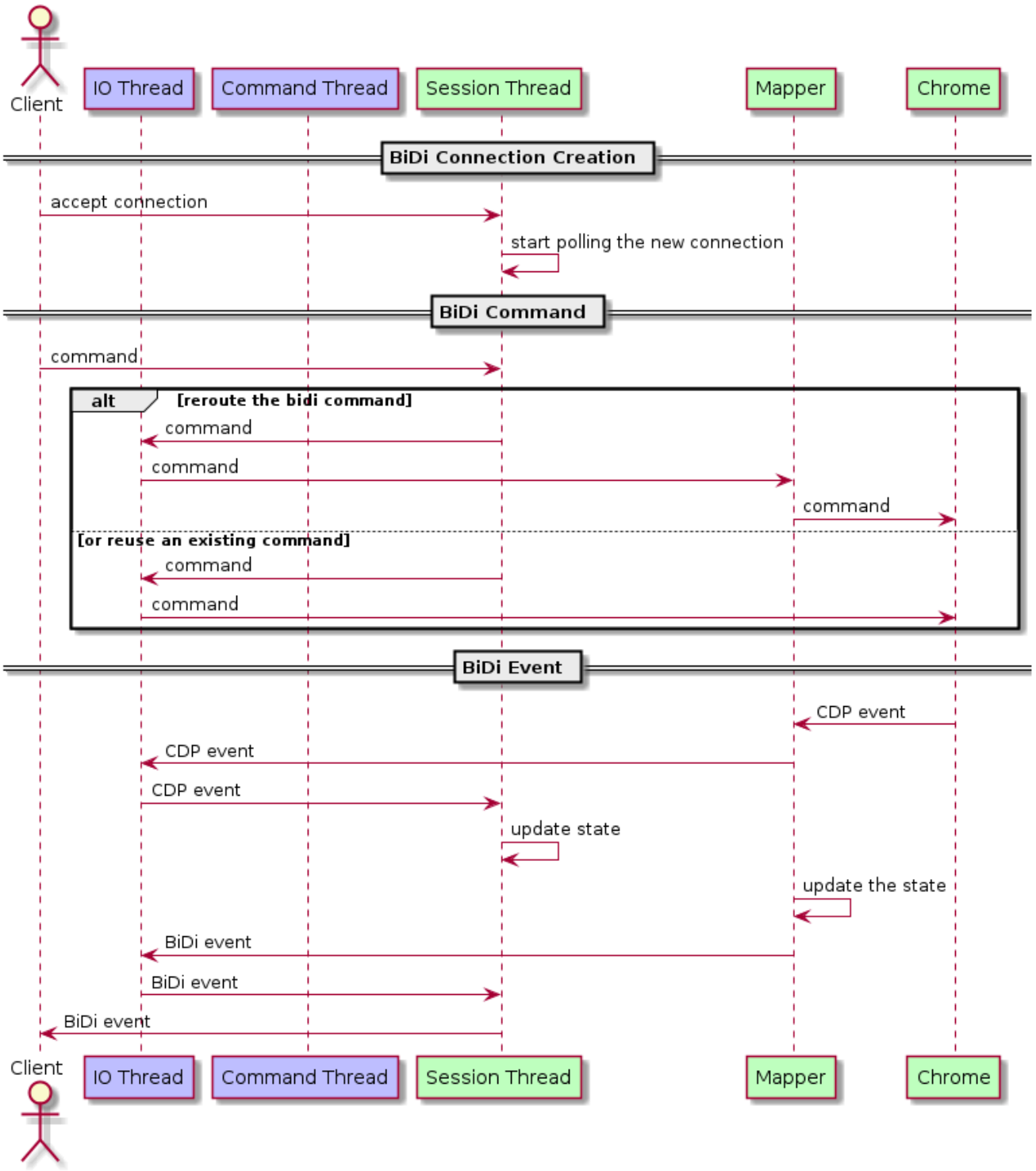
This means that Pure-BiDi has to be implemented differently. One idea for the implementation is described in the Alternative 4.

Alternative 2. Client communication in Session Thread

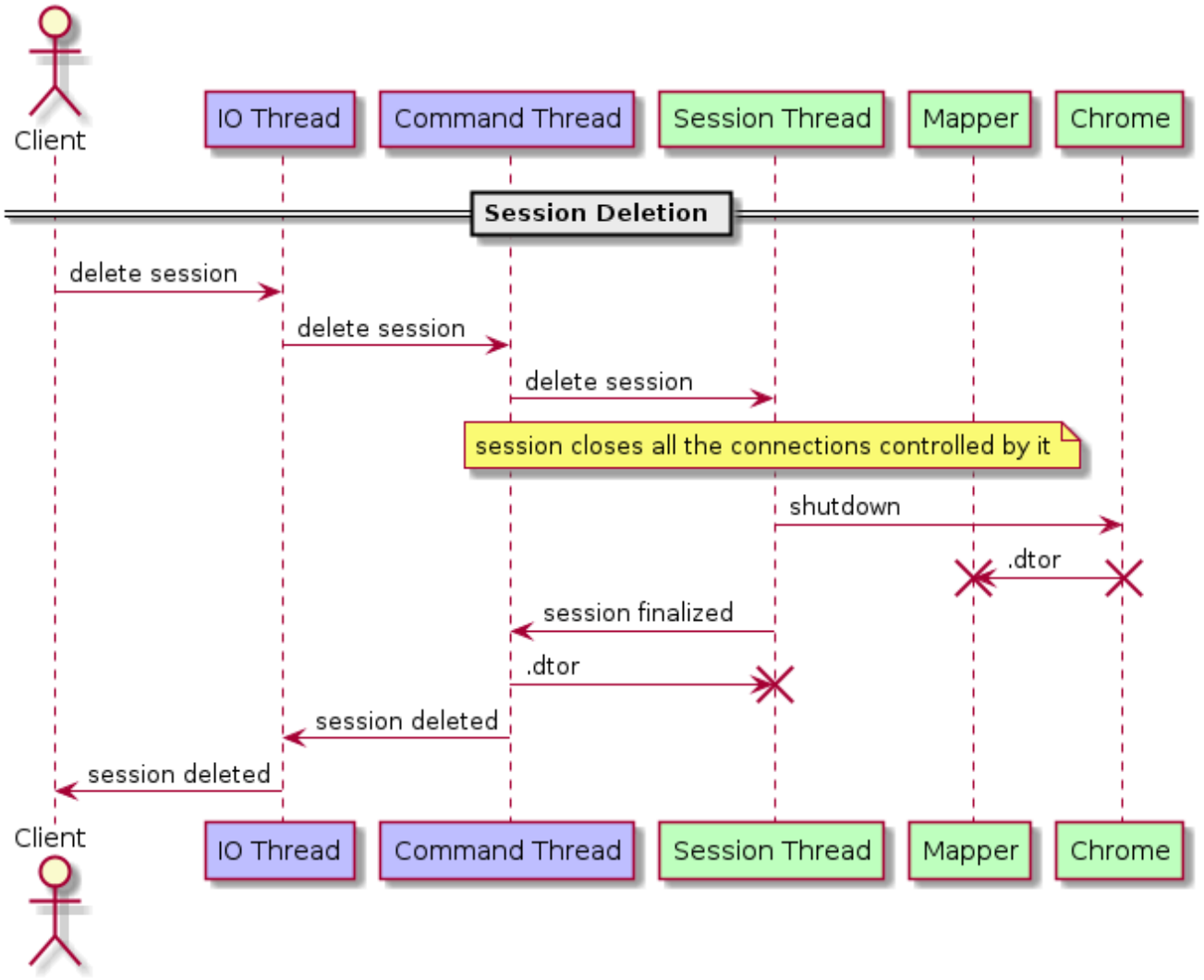
In this design we are merging Session Thread and Session IO Thread so that Session Thread handles both HTTP and BiDi traffic. In order to do that we need to change the message pump type of the session thread from Default to IO.



The user stories are handled as follows:



We do not create new CDP connections for each BiDi connection and therefore the traffic from multiple BiDi connections is routed through the single CDP connection. The session keeps information on what command has come through which channel and routes the response appropriately. The session also keeps the information about events to which each BiDi connection has subscribed so that BiDi events are routed by the session to the connections interested in these events.

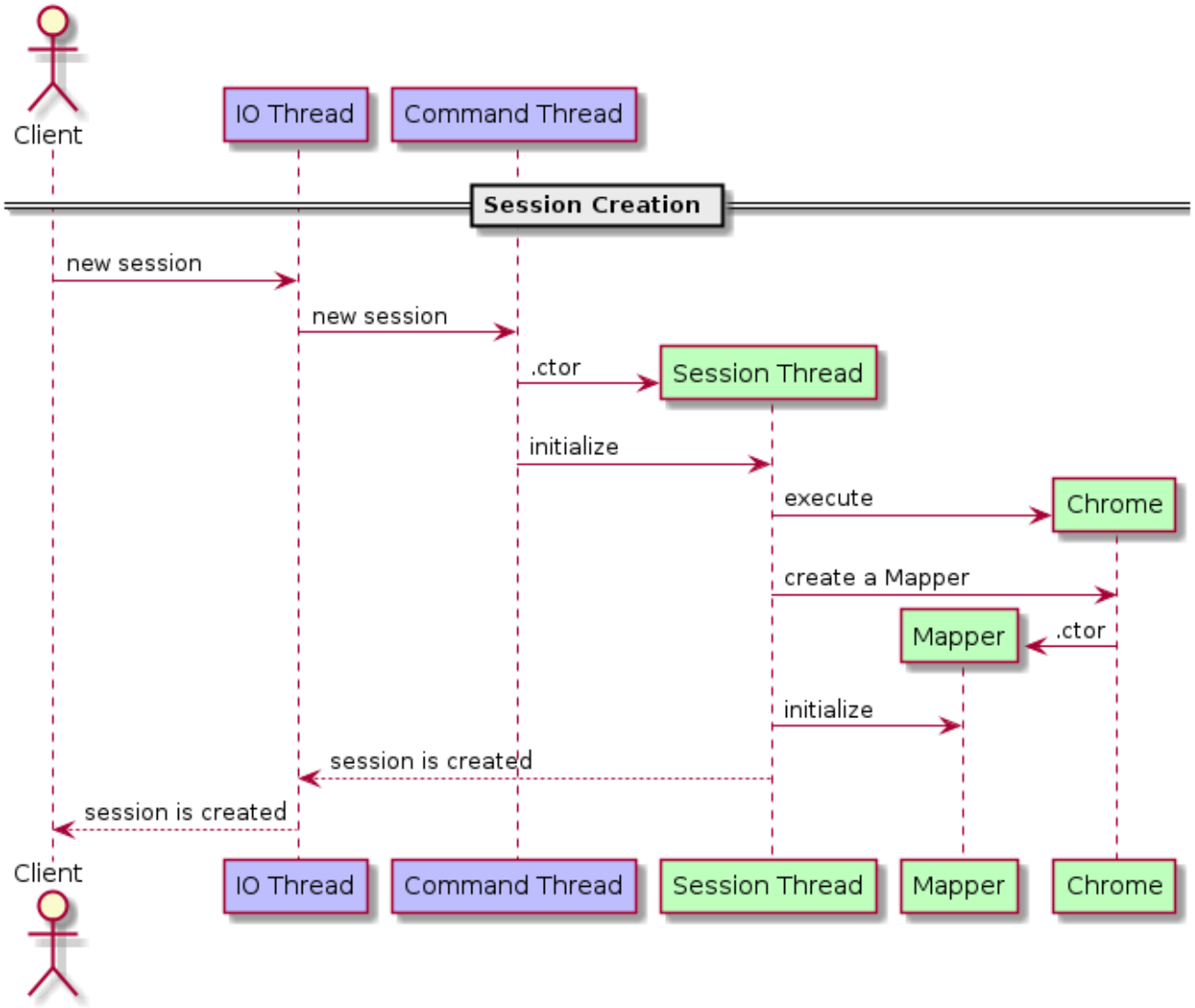


This design has the same problems as the Alternative 1.

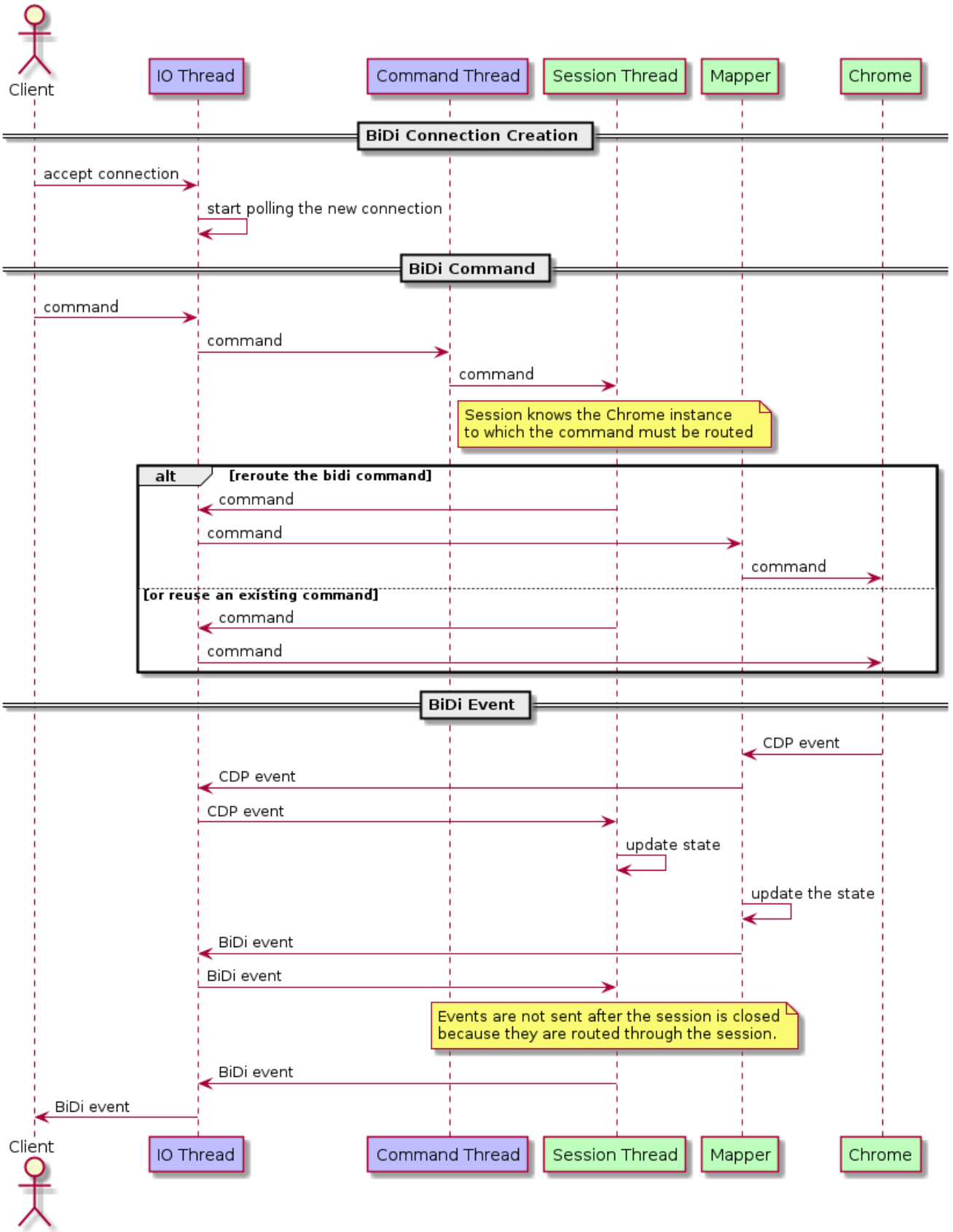
Alternative 3. Client communication in IO thread

In this design the IO thread carries out all communications.

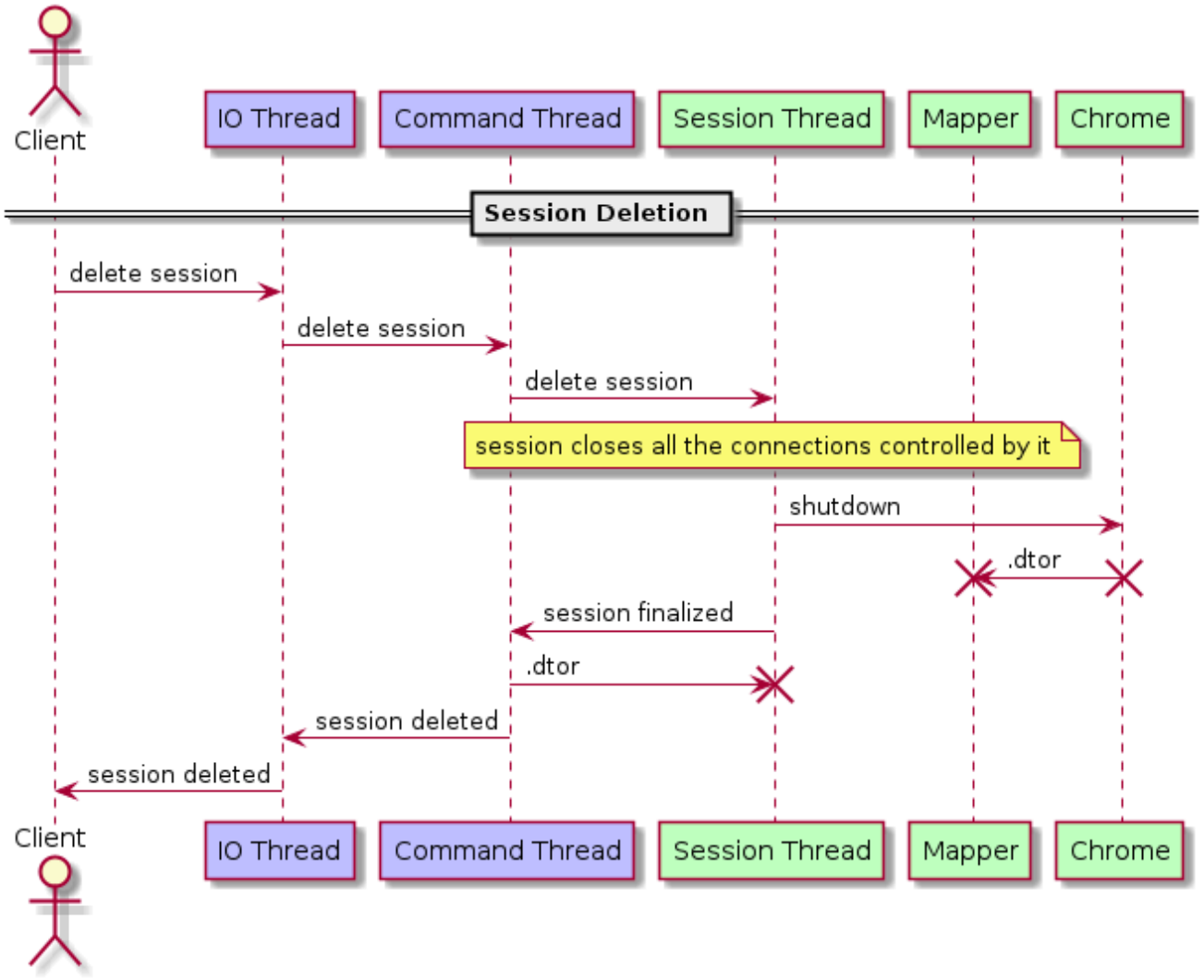
Session creation is identical to the Alternative 2.



The user stories are handled as follows:



Session deletion works the same way as in the current implementation.



The Pure-BiDi session can be implemented as follows:

- New connections are accepted on the same socket as the other WebSocket connections in the IO thread.
- The new connections not bound to any session are served by the CMD thread.
- As soon as the session.new command arrives via such an unbound connection a new HTTP-BiDi session is created. That is there is no such class as Pure-BiDi session.

Alternative 4. Single thread for BiDi + IO multiplexing

This design can be used to implement Pure-BiDi connections.

It is explained here at a very coarse level as we are unlikely to implement Pure-BiDi sessions now (2022Q2).

We have a single thread with a listener separate from the listener used by the HTTP requests handler.
All user connections are polled by the listener thread that accepts them.
New session requests are first sent to the Command Thread, where the new session is registered, and then handled by the listener thread.
These sessions are completely different from the classic sessions and essentially they are maps between the user and browser connections.
This design allows multiple Mappers and multiple CDP connections because Pure-BiDi sessions will

be stateless (not counting the aforementioned table).

Rollout plan

Waterfall.

As the WebDriver BiDi standard is currently in immature state and it supports too few commands the Pure-BiDi sessions will hardly be useful.
Most likely the users will prefer HTTP-BiDi sessions because they bring them the best from both worlds.
Therefore we will not work on Pure-BiDi sessions in the nearest future (2022Q2 or whole 2022).

Multiple connections per session will not be supported in 2022Q2 because they require a lot of bookkeeping to implement correct response and event routing.

Core principle considerations

Speed

Chrome performance is unaffected.

ChromeDriver performance must improve for ChromeDriver classic due to the active event consumption - a new user HTTP command will not have to wait until all the pending events are processed.

Chrome memory usage will increase under automation due to injection of Mapper code.

Security

As ChromeDriver and Chrome communicate over an unprotected channel we should be explicit to the users that TLS support by the ChromeDriver does not guarantee any safety of the traffic because it can be sniffed while being re-transmitted between ChromeDriver and Chrome.

Simplicity

<Describe how the feature fits into the rest of DevTools. Potential questions to answer:

- "Have I considered and documented unhappy paths or unwelcome results of these changes for users?"
- "Have I defined the user problems clearly and addressed them?"
- "Have I made sure all non-trivial work is done off the UI thread, so the UI is never unresponsive for > 200ms?" - UI jank is the devil. Don't make your change contribute to it.
- "Have I done my best to avoid introducing unexpected modal workflows, popups, questions the user can't answer, superfluous extra options?" - we bend over backwards to avoid these things.

- "Have I spent time before the next major release polishing my feature, making sure it has all appropriate animations, plays nicely with other Chrome features like themes, etc?"

If you are in doubt about the impact of your design for the user, followup with UX / PM first.>

Testing plan

We want to rely on integration tests written by us and WPT written by the community and by us.

Follow-up work

We need to think through the design for Pure-BiDi sessions - it might affect the design for HTTP-BiDi.

Open Questions

1. How are we going to close a Pure-BiDi session? There is no corresponding command in the standard. There is a ticket concerning this topic.
2. Are we going to allow the user to reconnect? How are we going to do that? There is a thread concerning the reconnection possibility.
3. What if the user issues a session.new command via the connection already bound to some session? It is not clear what we should do with the already existing session.
4. When we close a session what are we going to do with its connections? Shall we close them or make them unbound?
5. How can we ensure interoperability between BiDi session commands and HTTP session commands for HTTP-BiDi sessions? For instance ExecuteSwitchToFrame supplies frames with property 'cd_frame_id_' thus providing an additional way of their identification. This id is used for example for bringing an element into a view while clicking it. We need to ensure that BiDi Mapper uses the same mechanisms as ChromeDriver Classic.
   Possible solution: execute this sort of commands via the means of classic session, don't handle them in the Mapper.