SEPTEMBER 28, 2021

NETTY ONLINE MEETUP

AGENDA

- Netty 5 alpha release plans (~ 5 mins)
- Netty 5 monthly online meetup (Poll) (~ 5 mins)
- HTTP/2 API Proposal (~ 40 mins)
- Q & A (~ 10 mins)

NETTY 5 RELEASE

- Alpha planned for February 2022

NETTY 5 RELEASE

- Alpha planned for February 2022
  - Important bug fixes only to 4.1

NETTY 5 RELEASE

- Alpha planned for February 2022
  - Important bug fixes only to 4.1
  - Help frameworks (netty based) maintainers to port to 5.x.

ROUTINE MEETUP?

- Significant movement on Netty 5.
  - Hard to track/discuss larger changes on Github.
  - Any interest in having a meetup on routine cadence?
    - Monthly/bimonthly?

HTTP/2 API CHANGES

WHY?

- Netty 4 HTTP/2 API challenges

WHY?

- Netty 4 HTTP/2 API challenges
    - Two distinct API approaches, causes confusion.

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.
- Other insights

HTTP/2 API CHANGES

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.
- Other insights
  - HTTP/3 learnings

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.
- Other insights
  - HTTP/3 learnings
  - Child channel performance overhead

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.
- Other insights
  - HTTP/3 learnings
  - Child channel performance overhead
  - Control frame visibility

WHY?

- Netty 4 HTTP/2 API challenges
  - Two distinct API approaches, causes confusion.
  - Leaky abstraction b/w the API approaches
    - Flow control handling in child channel exposes connection API.
- Other insights
  - HTTP/3 learnings
  - Child channel performance overhead
  - Control frame visibility
  - Auto-magical details for close and go_away coupling

GOALS

- Propose a single API with a channel instance per HTTP/2 stream

GOALS

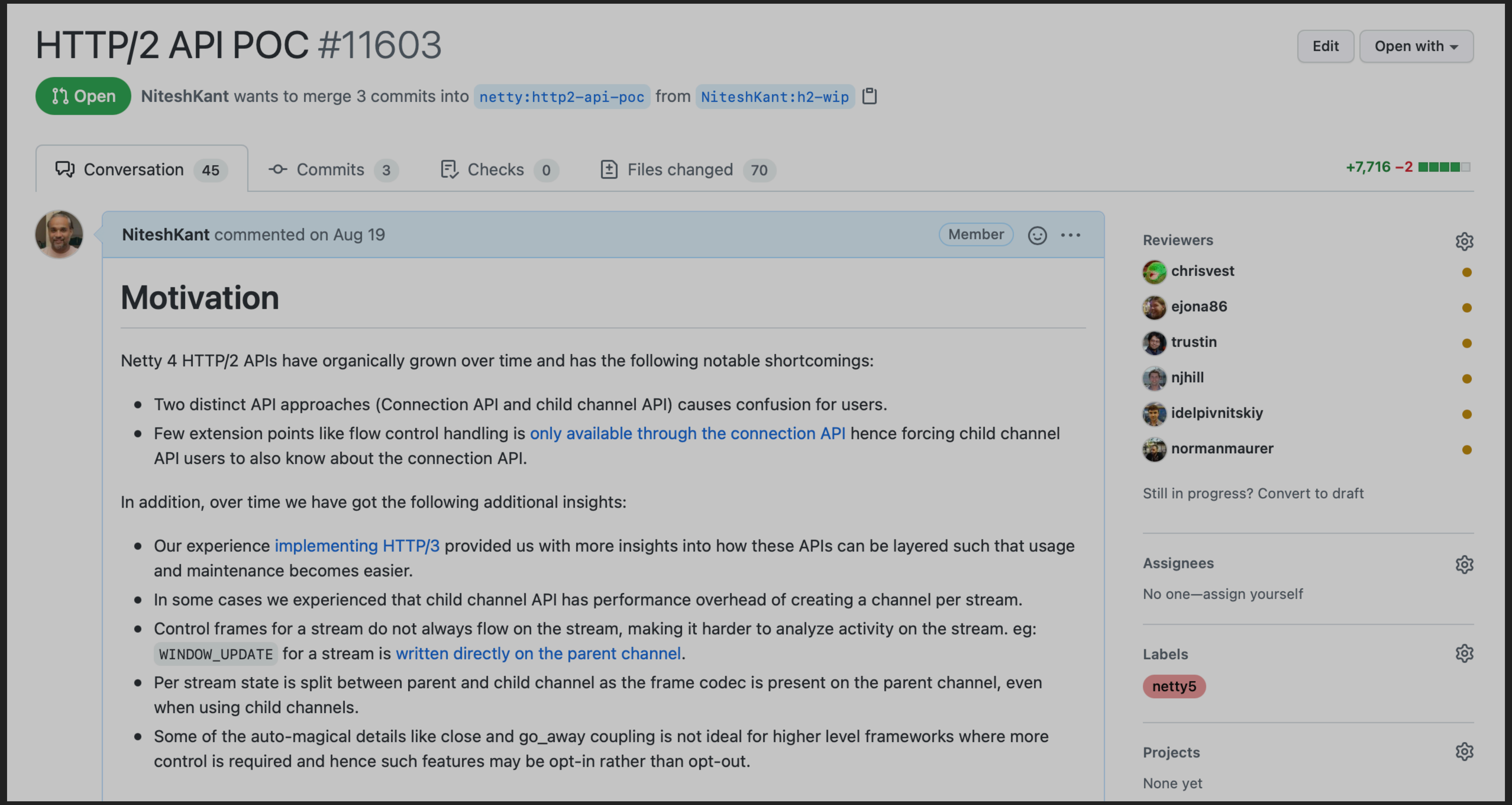
- Propose a single API with a channel instance per HTTP/2 stream
- *Experiment* on reducing child channel performance overhead

NON-GOALS

- Backward compatibility with 4.1

PROPOSAL

https://github.com/netty/netty/pull/11603



HTTP/2 API POC #11603

Edit | Open with

Open | NiteshKant wants to merge 3 commits into `netty:http2-api-poc` from `NiteshKant:h2-wip`

Conversation 45 | Commits 3 | Checks 0 | Files changed 70 | +7,716 −2

NiteshKant commented on Aug 19 | Member

Motivation

Netty 4 HTTP/2 APIs have organically grown over time and has the following notable shortcomings:

- Two distinct API approaches (Connection API and child channel API) causes confusion for users.
- Few extension points like flow control handling is only available through the connection API hence forcing child channel API users to also know about the connection API.

In addition, over time we have got the following additional insights:

- Our experience implementing HTTP/3 provided us with more insights into how these APIs can be layered such that usage and maintenance becomes easier.
- In some cases we experienced that child channel API has performance overhead of creating a channel per stream.
- Control frames for a stream do not always flow on the stream, making it harder to analyze activity on the stream. eg: `WINDOW_UPDATE` for a stream is written directly on the parent channel.
- Per stream state is split between parent and child channel as the frame codec is present on the parent channel, even when using child channels.
- Some of the auto-magical details like close and go_away coupling is not ideal for higher level frameworks where more control is required and hence such features may be opt-in rather than opt-out.

Reviewers: chrisvest, ejona86, trustin, njhill, idelpivnitskiy, normanmaurer

Still in progress? Convert to draft

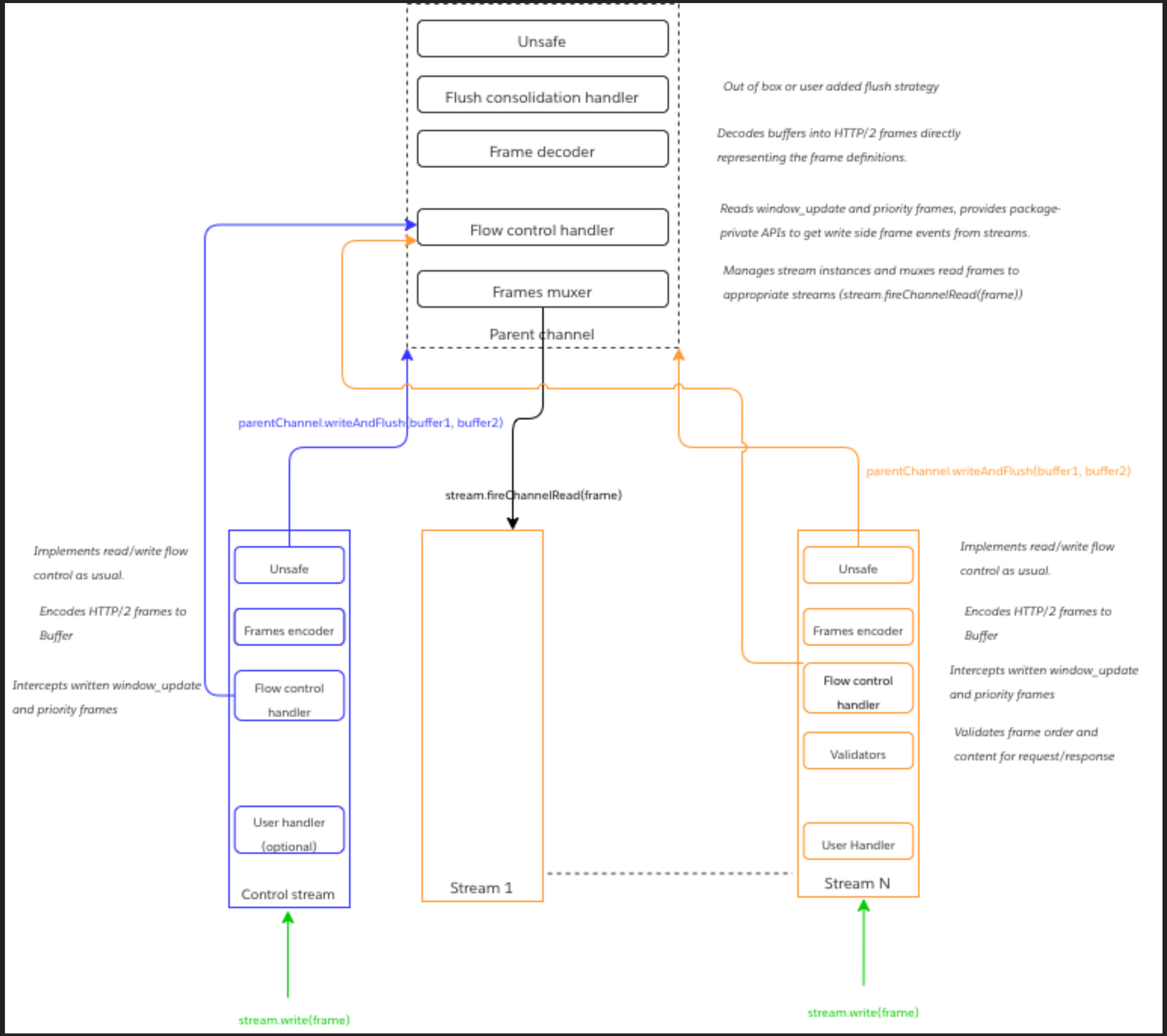
Assignees: No one—assign yourself

Labels: netty5

Projects: None yet

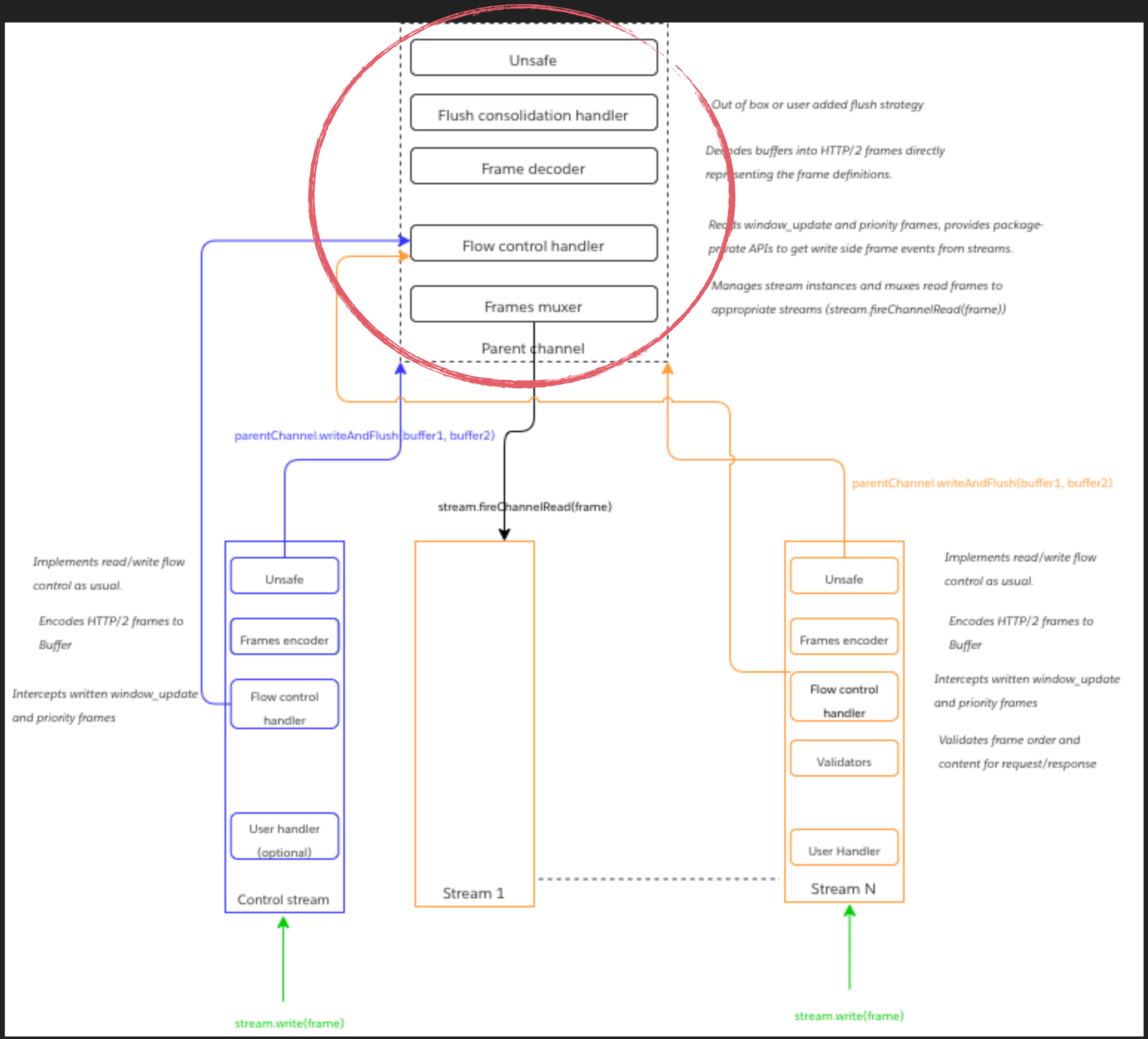
CHANNEL SETUP

CHANNEL SETUP


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CHANNEL SETUP

Parent channel



Out of box or user added flush strategy

Decodes buffers into HTTP/2 frames directly representing the frame definitions.

Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.

Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))

Implements read/write flow control as usual.

Encodes HTTP/2 frames to Buffer

Intercepts written window_update and priority frames

Implements read/write flow control as usual.

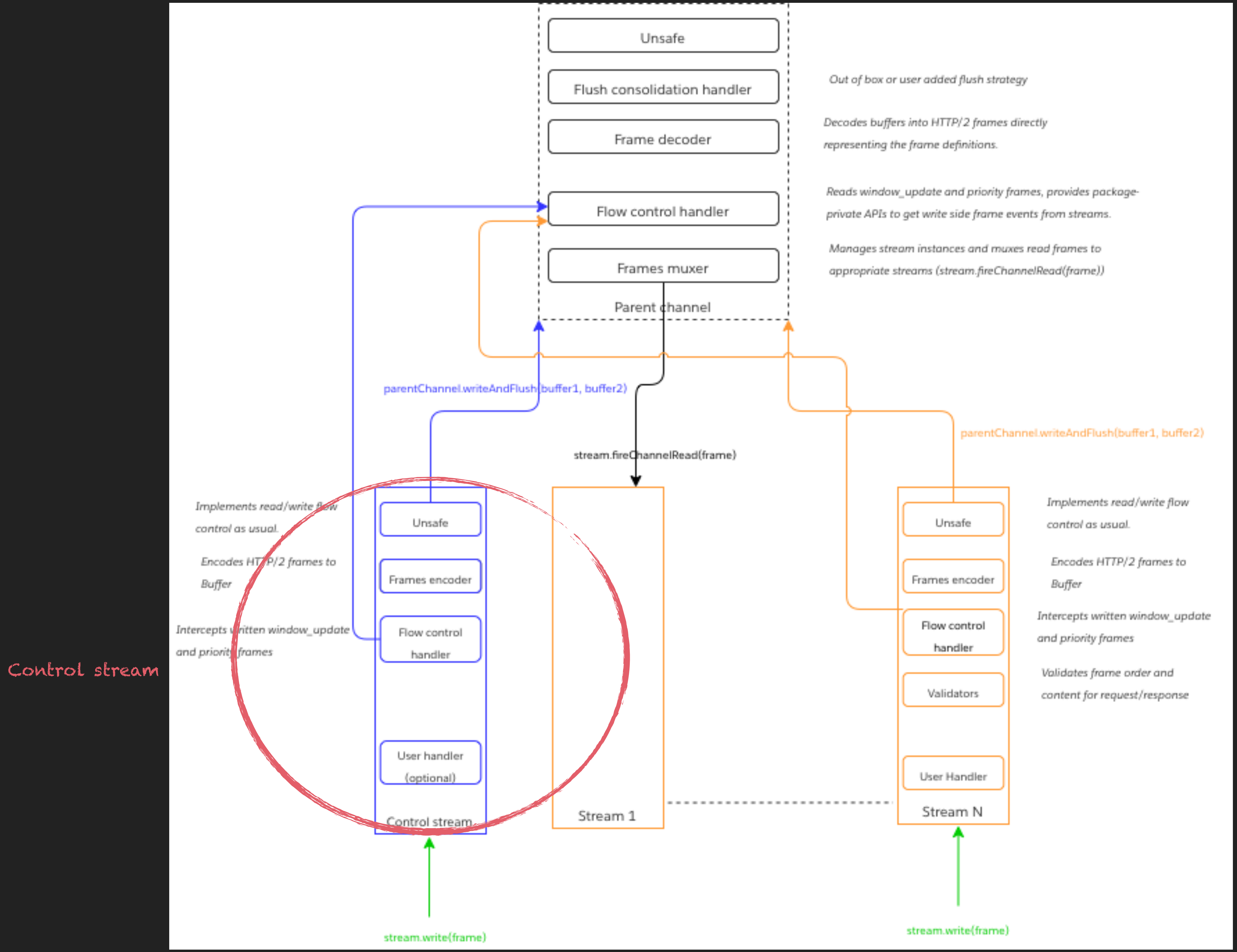
Encodes HTTP/2 frames to Buffer

Intercepts written window_update and priority frames

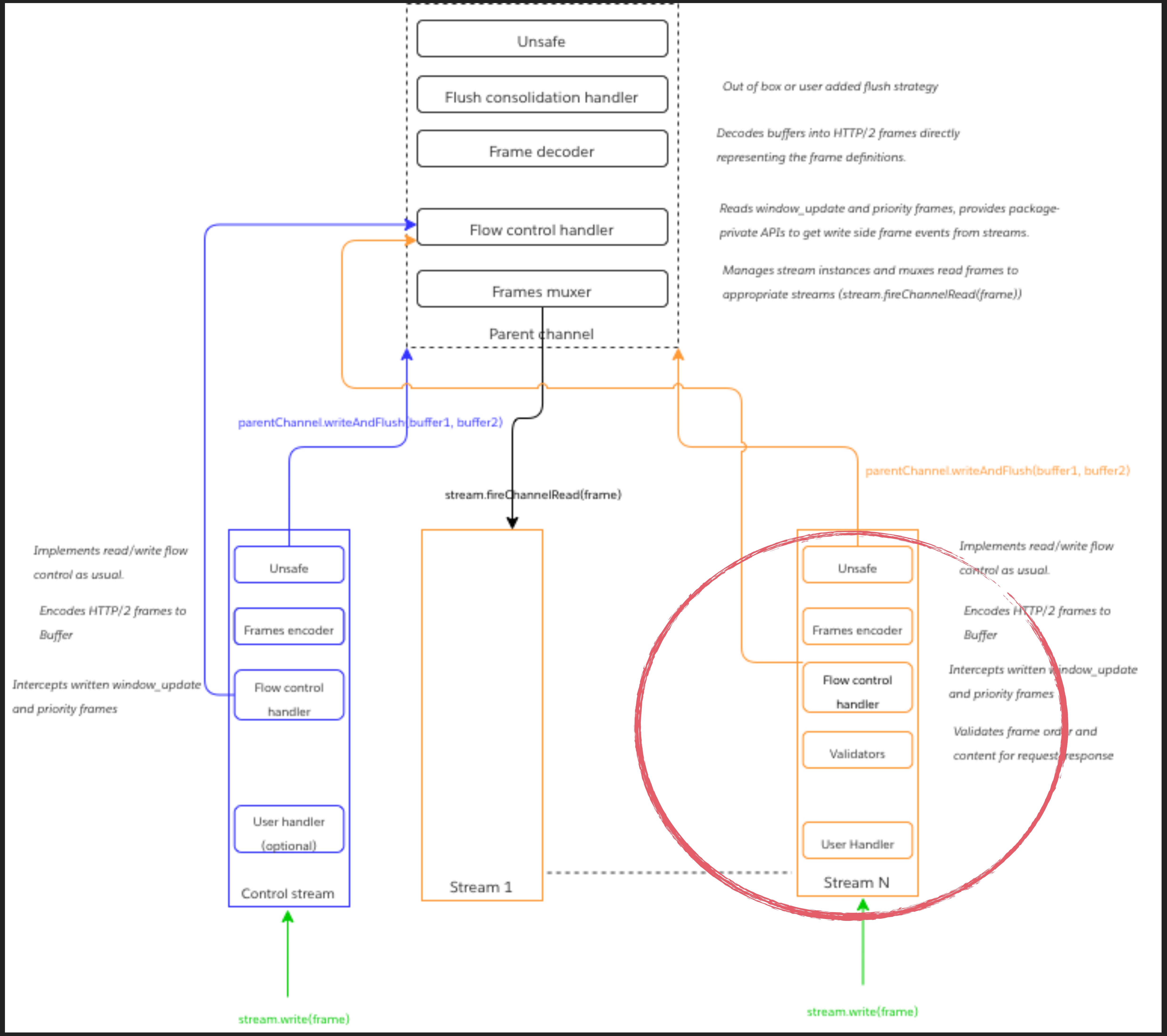
Validates frame order and content for request/response

HTTP/2 API CHANGES

CHANNEL SETUP


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Control stream
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

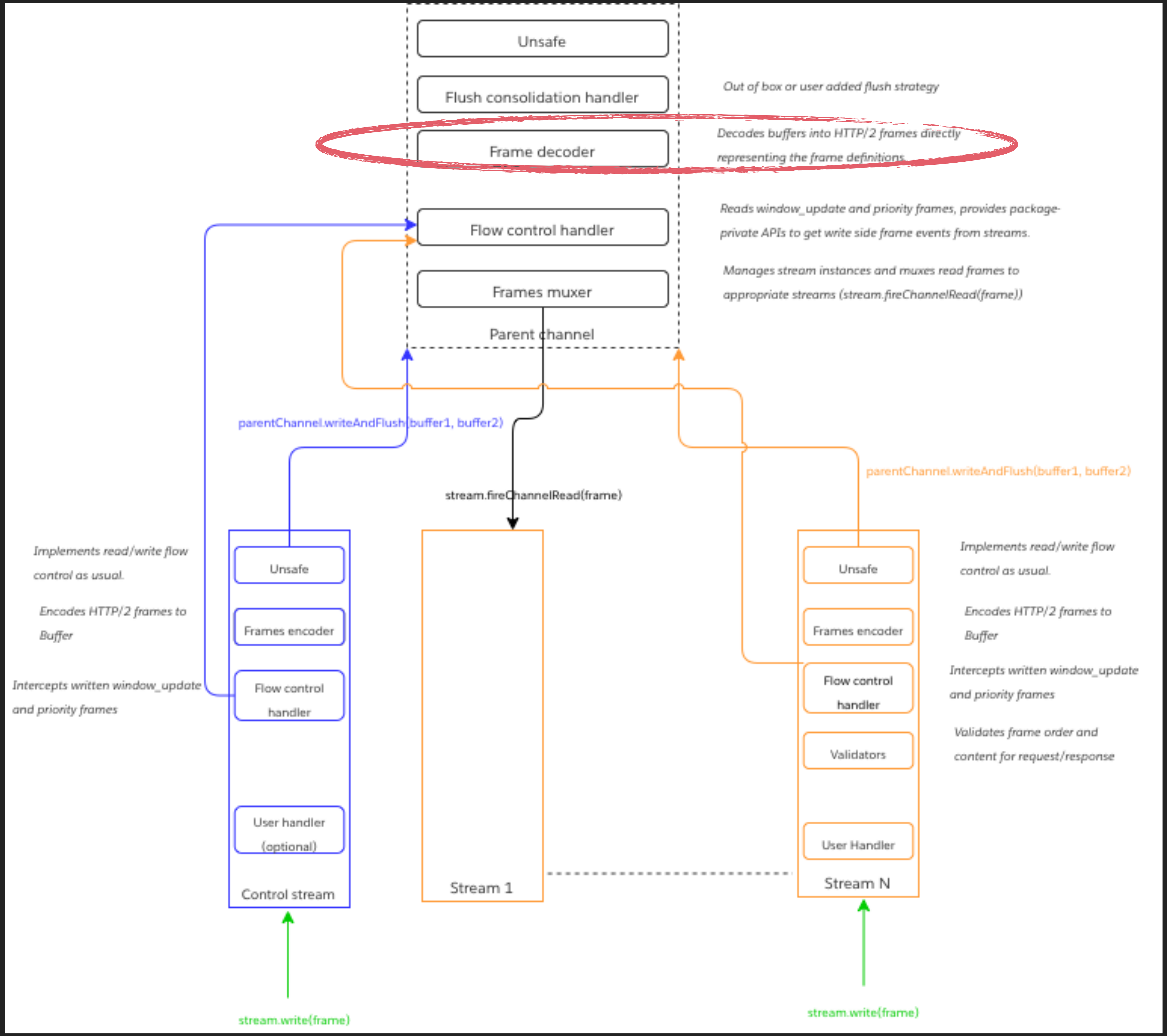
CHANNEL SETUP


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

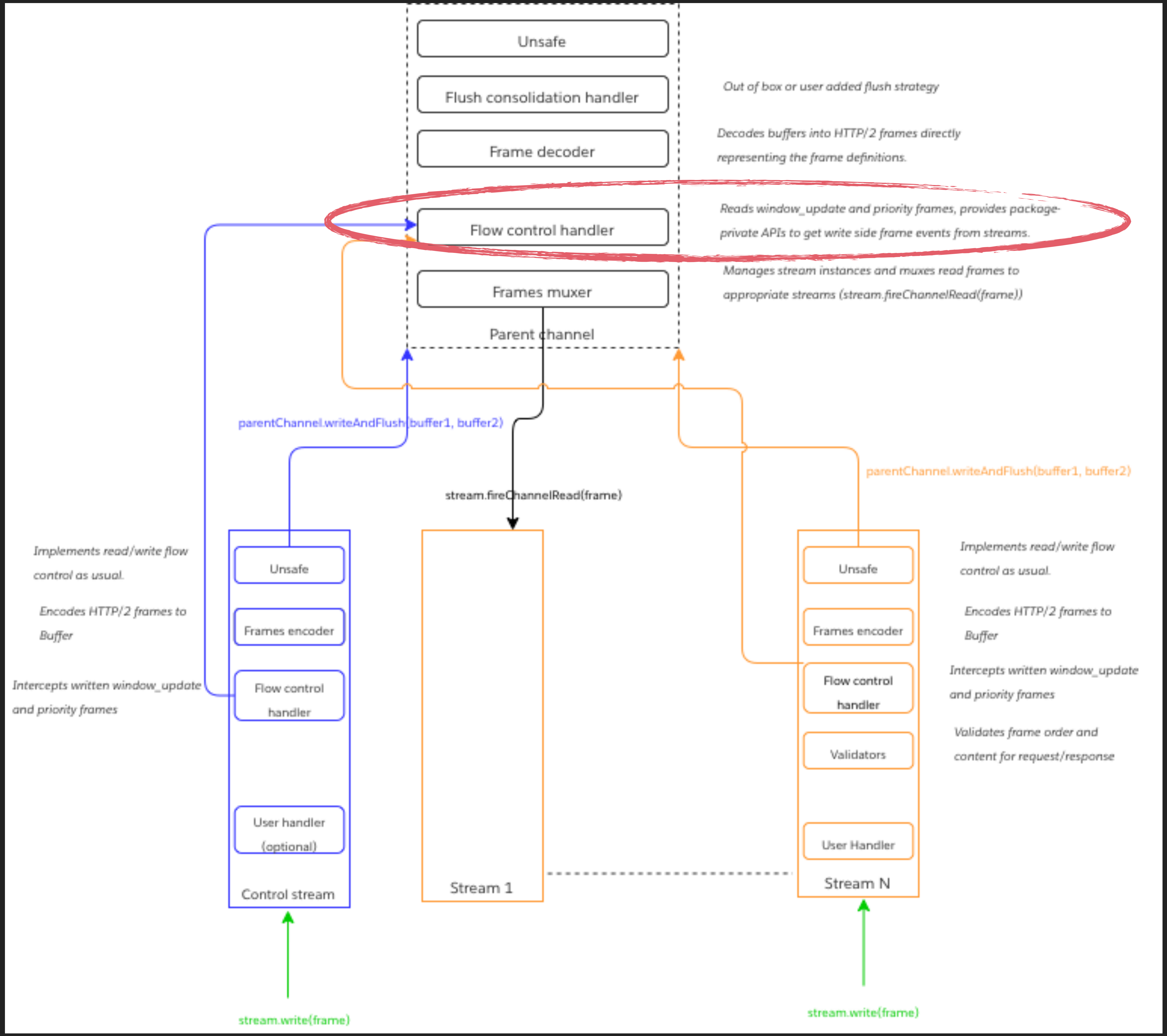
Request/Response stream

CONTROL FLOW (READ)

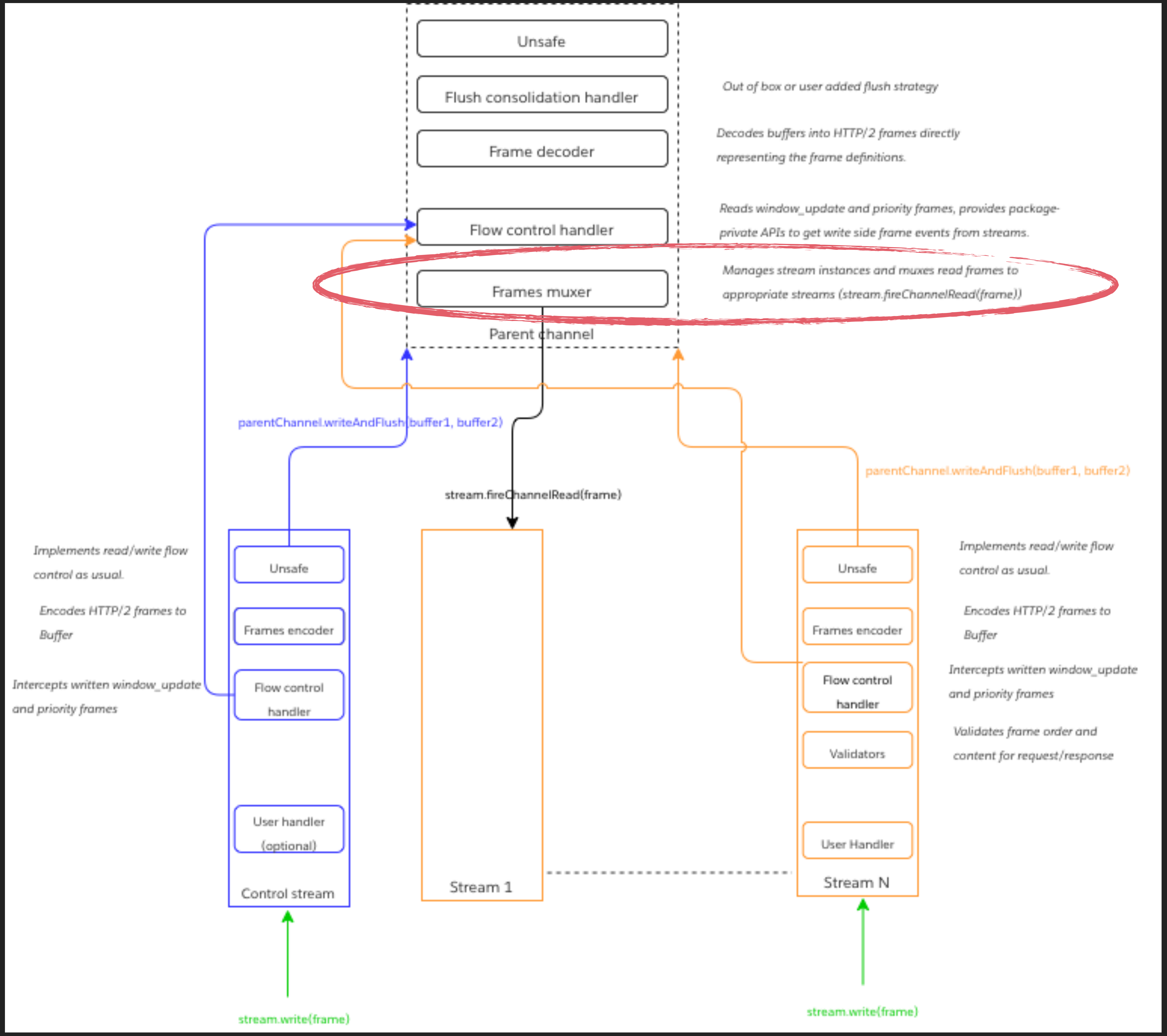
CONTROL FLOW (READ)


Unsafe
Flush consolidation handler
Out of box or user added flush strategy
Frame decoder
Decodes buffers into HTTP/2 frames directly representing the frame definitions
Flow control handler
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Frames muxer
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
Parent channel
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

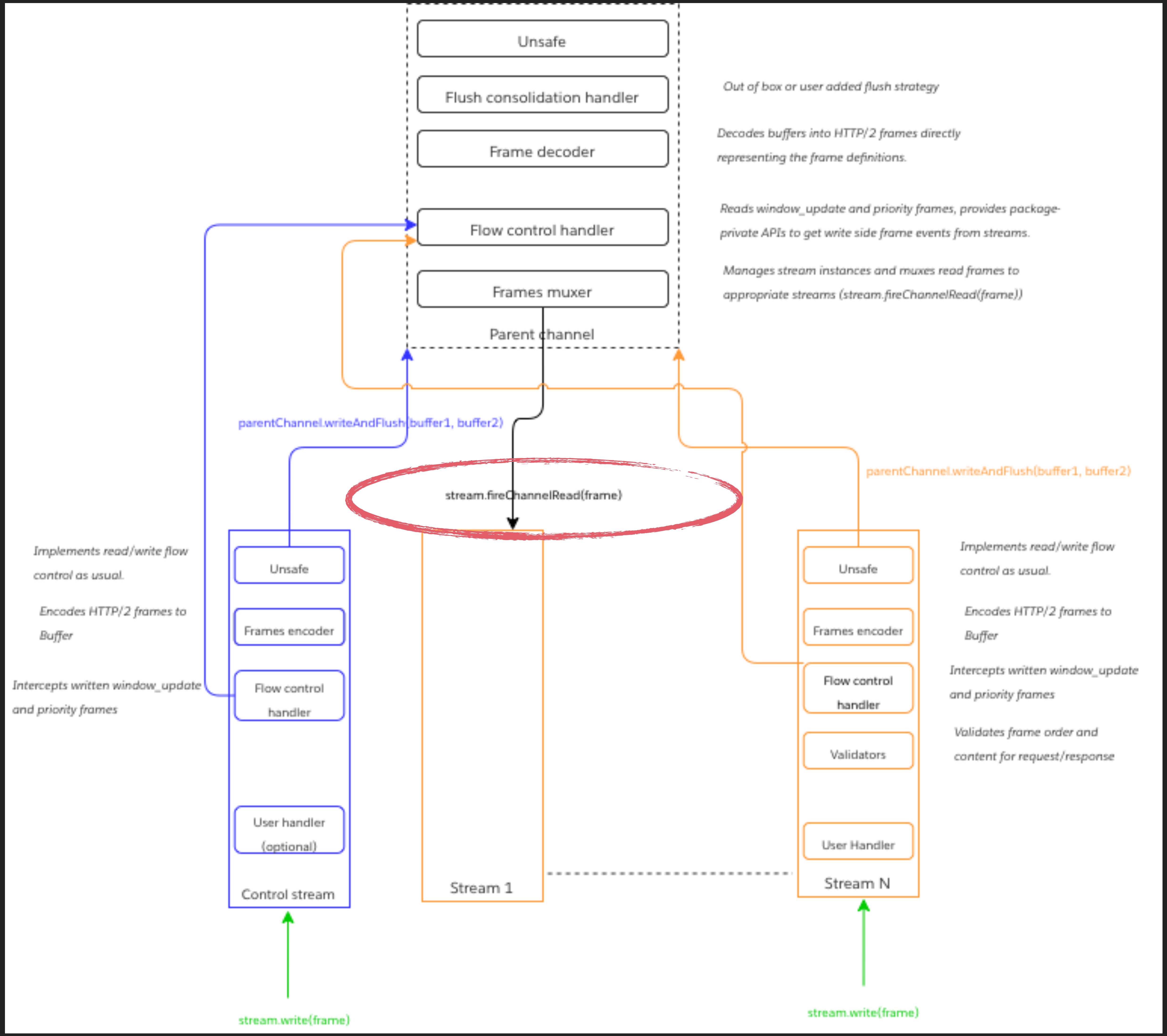
CONTROL FLOW (READ)


Unsafe
Flush consolidation handler
Out of box or user added flush strategy
Frame decoder
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Flow control handler
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Frames muxer
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
Parent channel
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (READ)


Unsafe
Flush consolidation handler
Out of box or user added flush strategy
Frame decoder
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Flow control handler
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Frames muxer
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
Parent channel
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (READ)



Out of box or user added flush strategy

Decodes buffers into HTTP/2 frames directly representing the frame definitions.

Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.

Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))

Implements read/write flow control as usual.

Encodes HTTP/2 frames to Buffer

Intercepts written window_update and priority frames

Implements read/write flow control as usual.

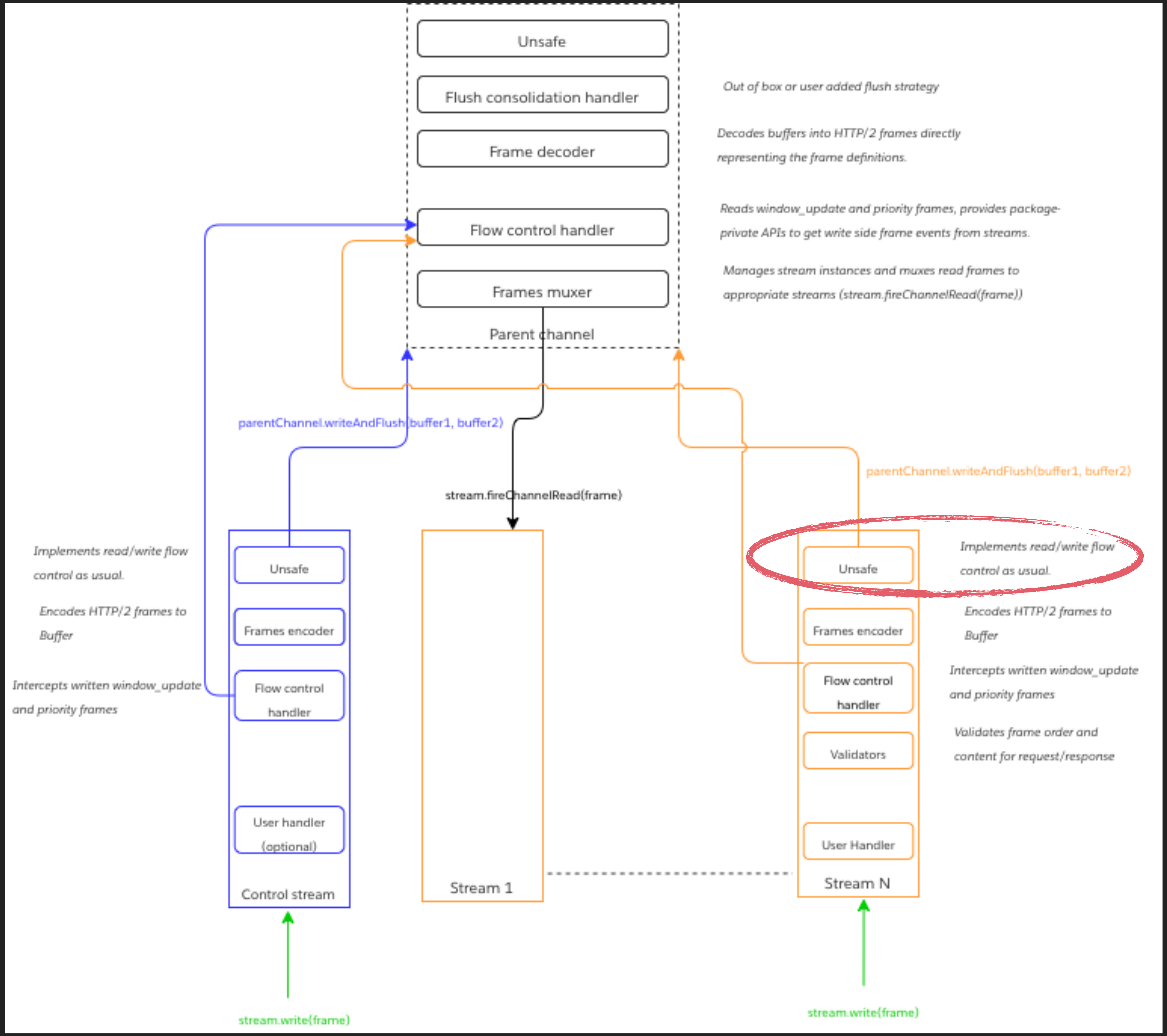
Encodes HTTP/2 frames to Buffer

Intercepts written window_update and priority frames

Validates frame order and content for request/response

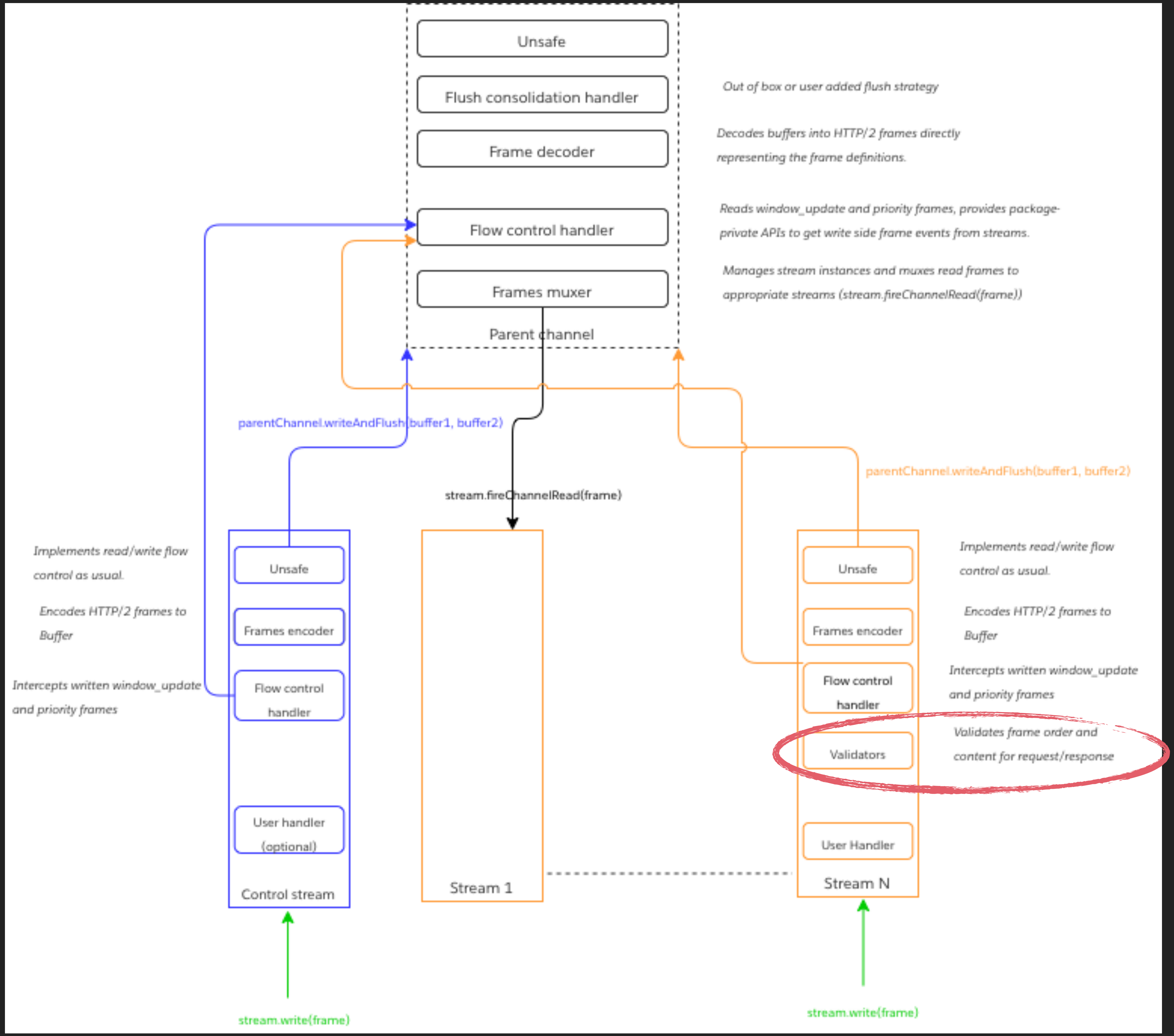
HTTP/2 API CHANGES

CONTROL FLOW (READ)

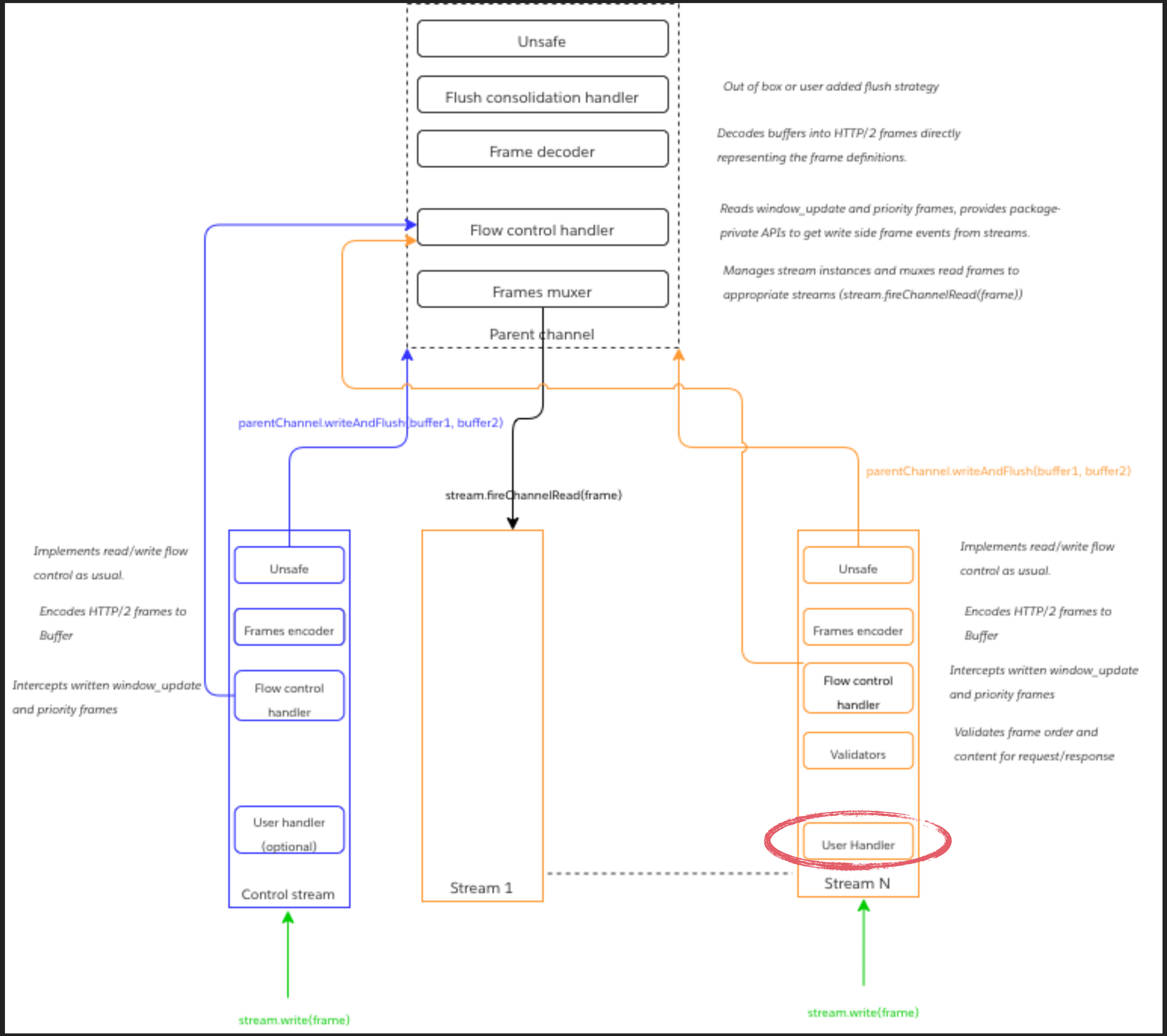

Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

Buffer reads based on netty flow control signals

CONTROL FLOW (READ)

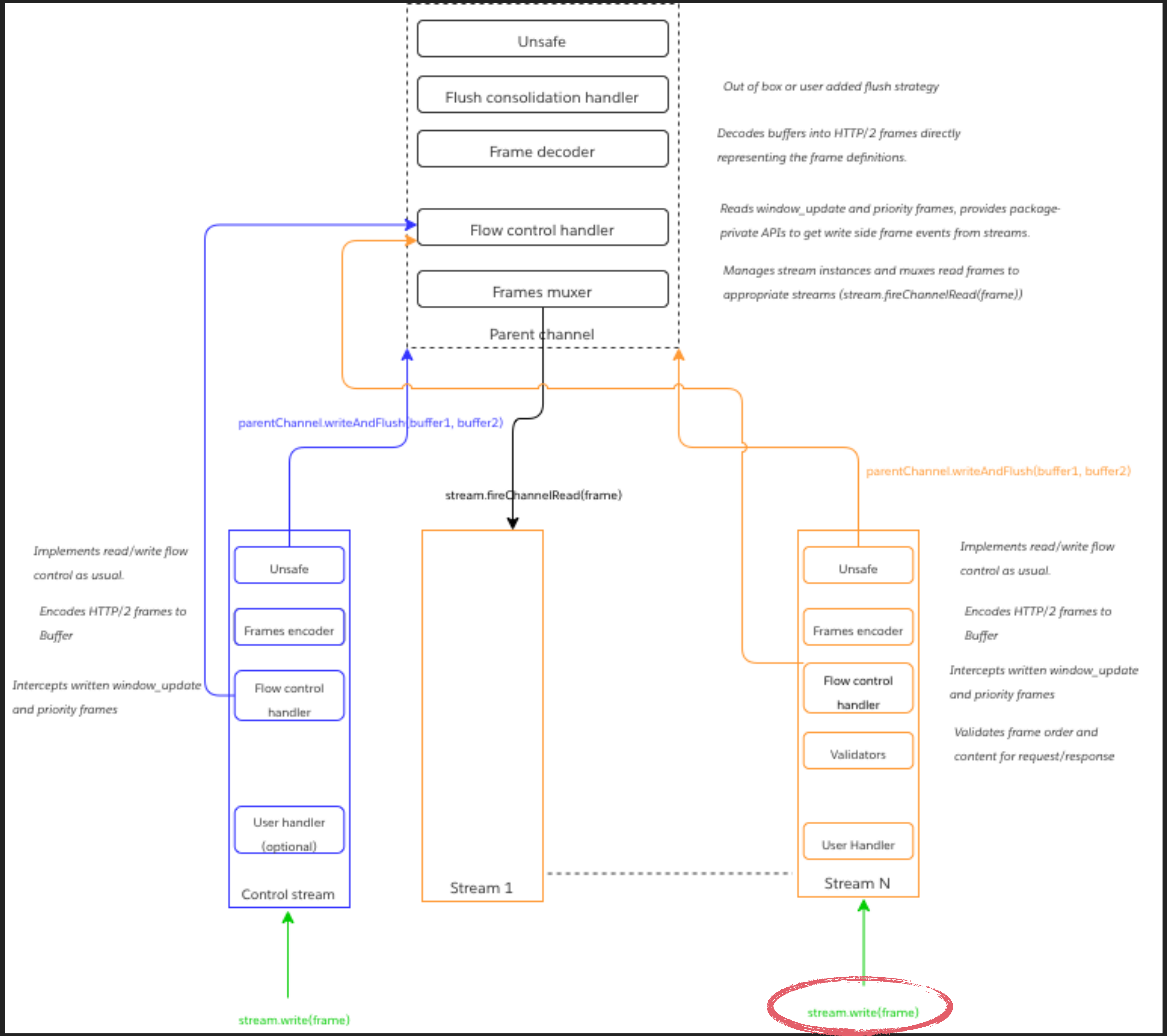

Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (READ)


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

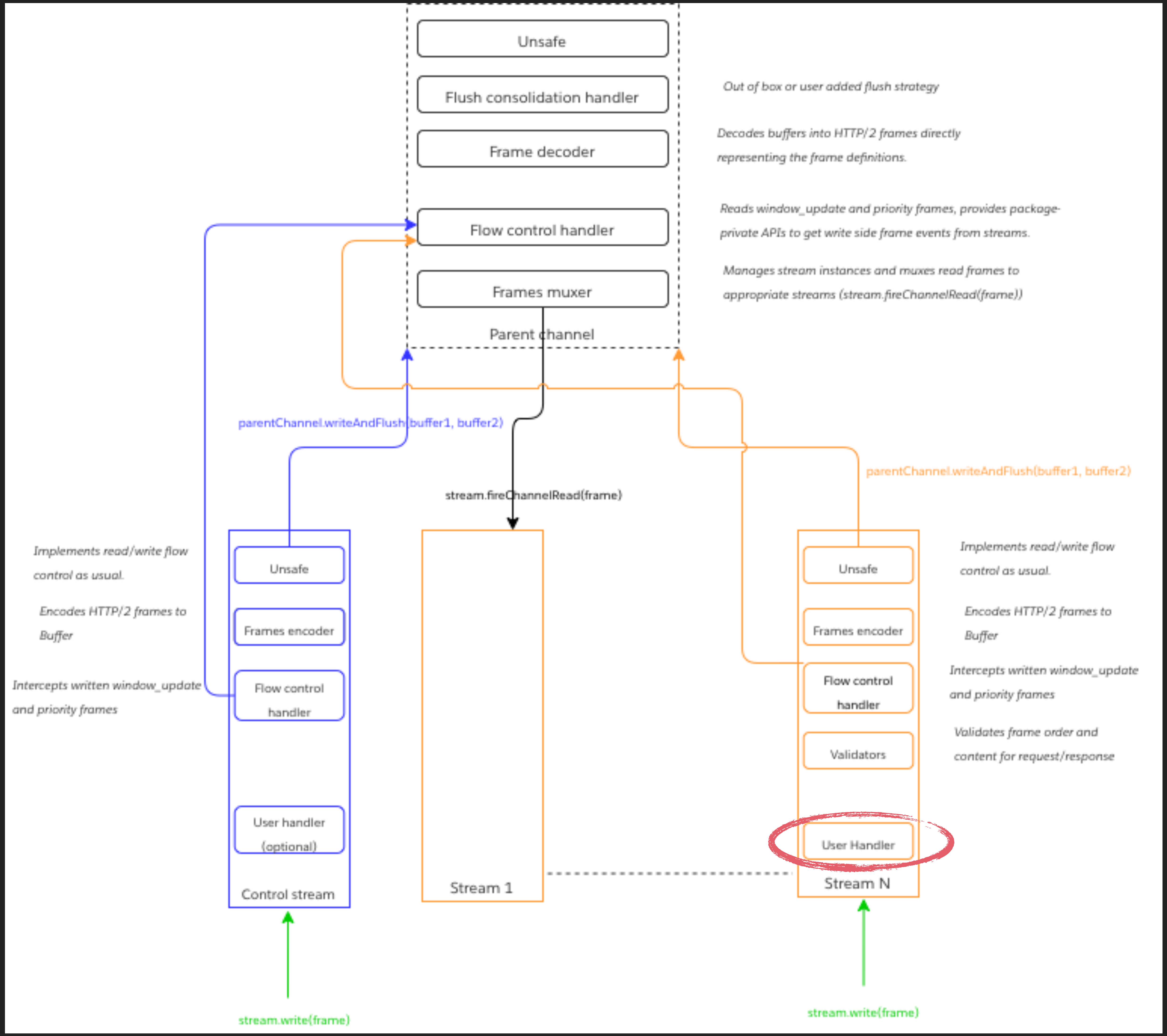
CONTROL FLOW(WRITE)

CONTROL FLOW (WRITE)

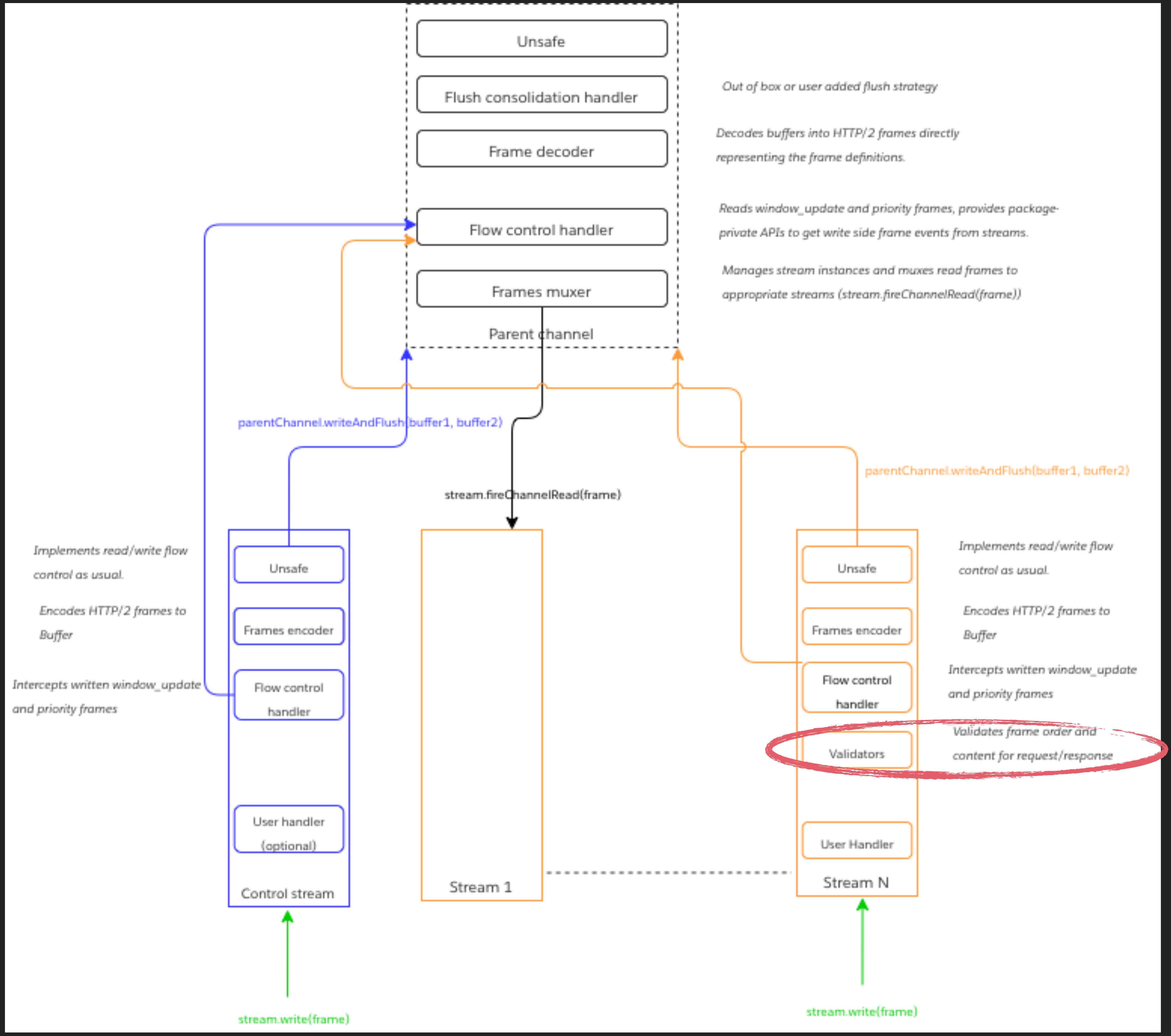

Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

User writes on a channel instance

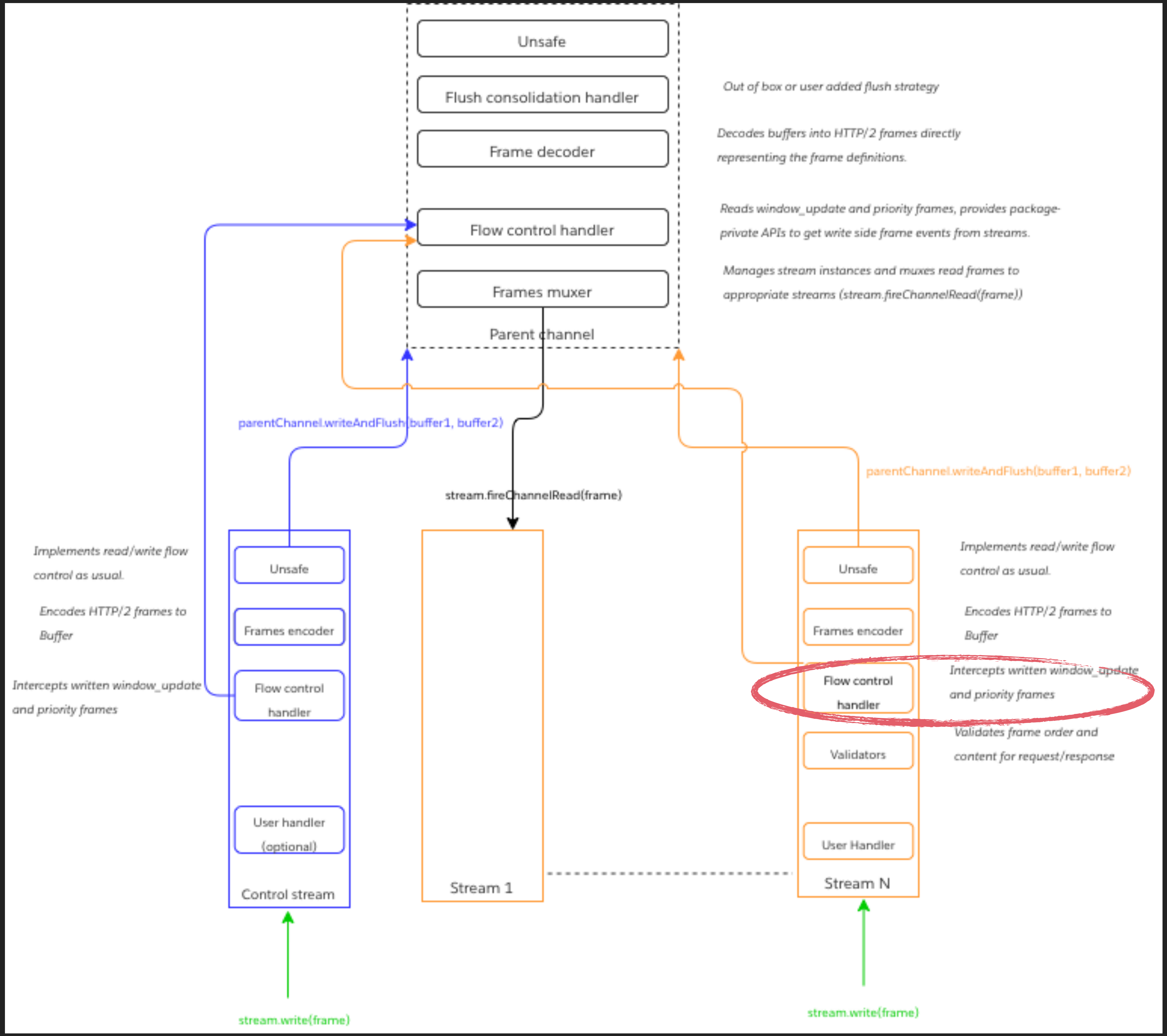
CONTROL FLOW (WRITE)


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (WRITE)


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (WRITE)

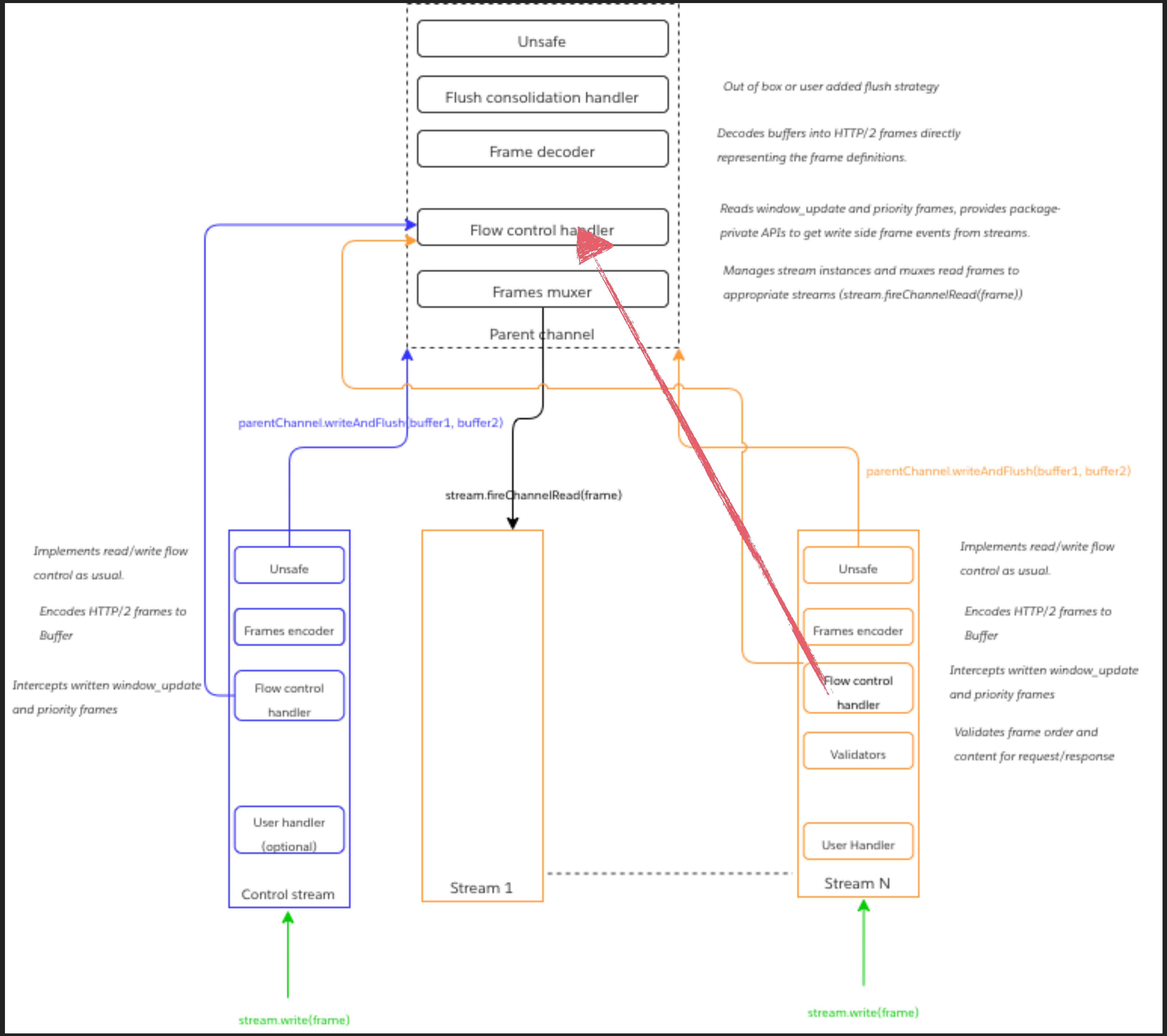

Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

Buffers writes

according to

flow control window sizes

CONTROL FLOW (WRITE)

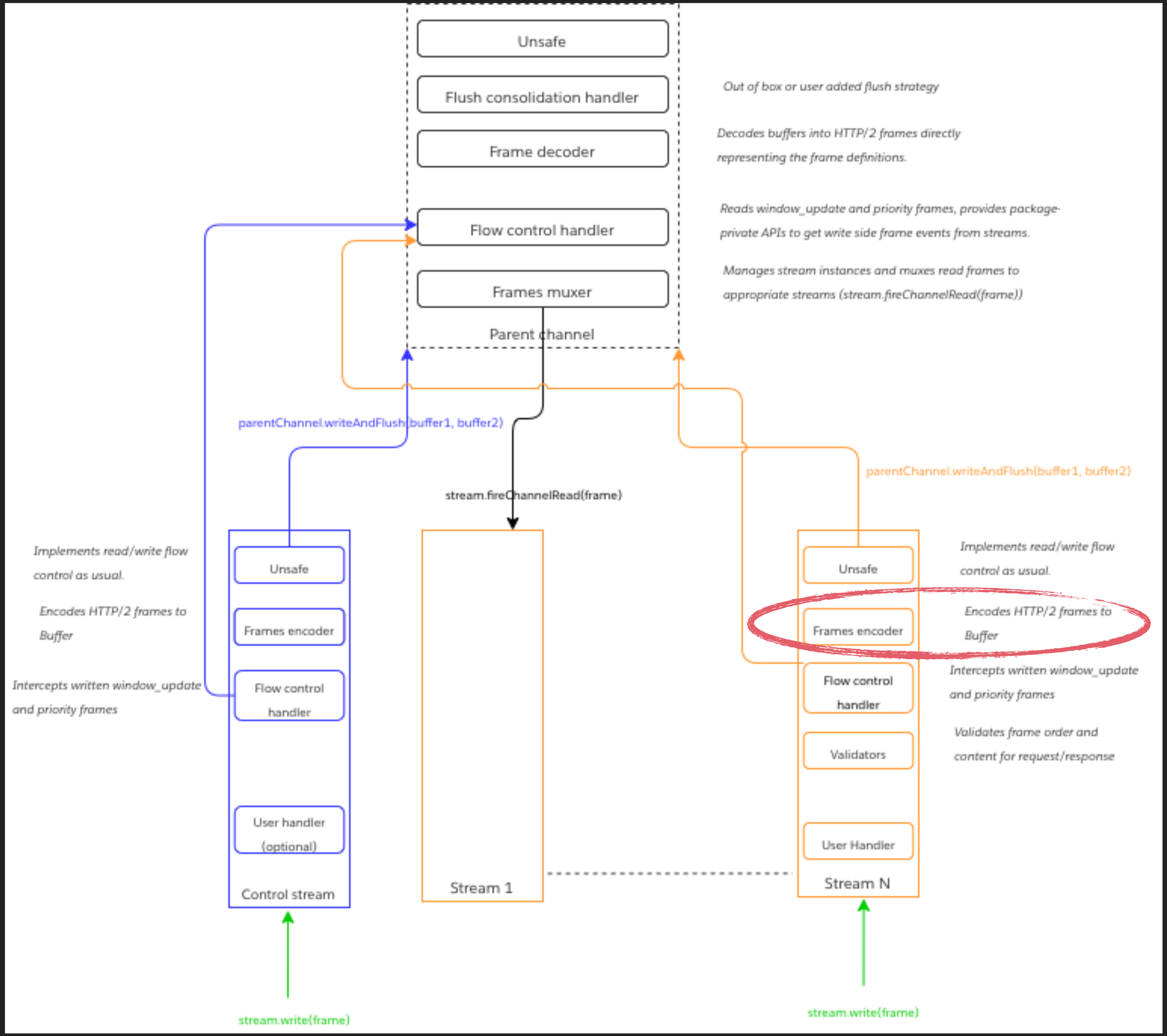

Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

Package private methods to aid

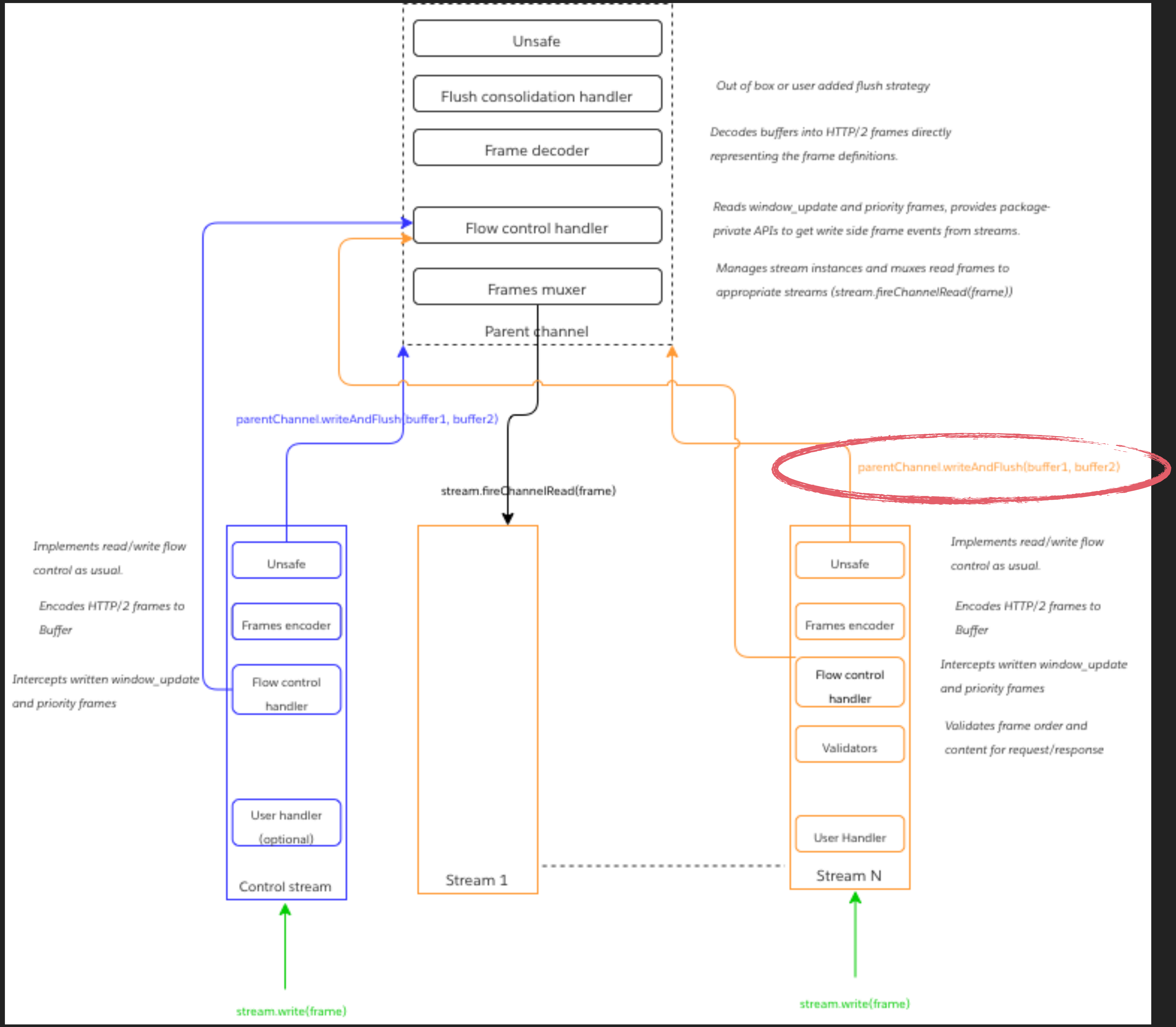
channel -> parent communications

HTTP/2 API CHANGES

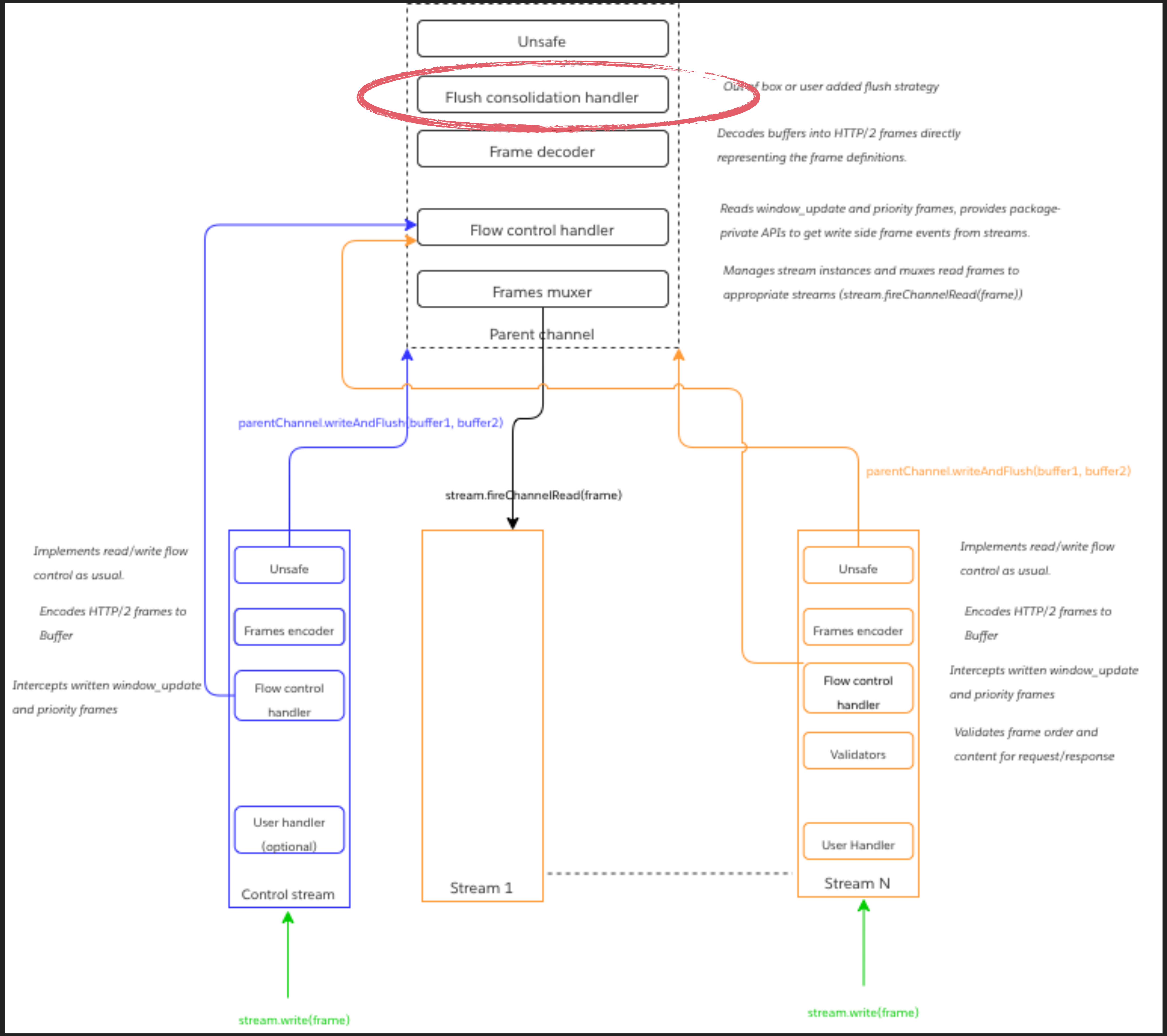
CONTROL FLOW (WRITE)


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

CONTROL FLOW (WRITE)


Unsafe
Flush consolidation handler
Frame decoder
Flow control handler
Frames muxer
Parent channel
Out of box or user added flush strategy
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

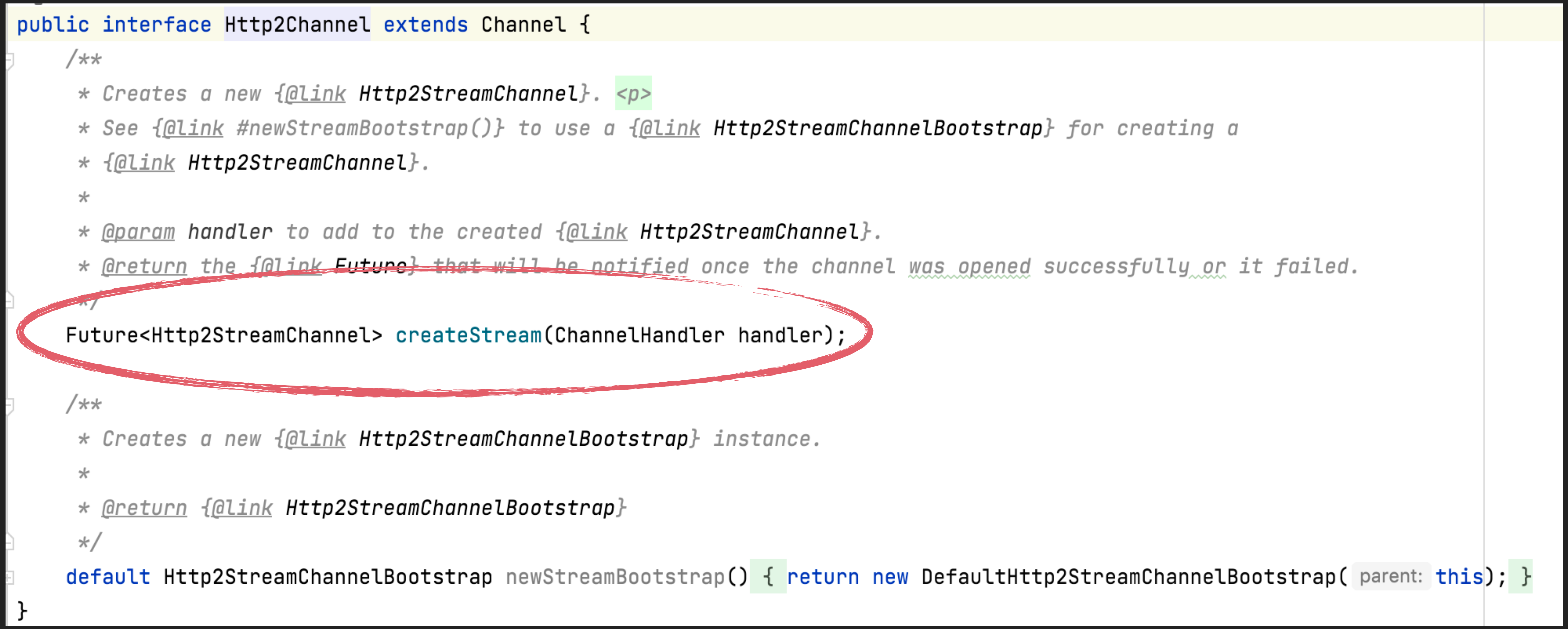
CONTROL FLOW (WRITE)


Unsafe
Flush consolidation handler
Out of box or user added flush strategy
Frame decoder
Decodes buffers into HTTP/2 frames directly representing the frame definitions.
Flow control handler
Reads window_update and priority frames, provides package-private APIs to get write side frame events from streams.
Frames muxer
Manages stream instances and muxes read frames to appropriate streams (stream.fireChannelRead(frame))
Parent channel
parentChannel.writeAndFlush(buffer1, buffer2)
stream.fireChannelRead(frame)
parentChannel.writeAndFlush(buffer1, buffer2)
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Unsafe
Frames encoder
Flow control handler
User handler (optional)
Control stream
Stream 1
Unsafe
Frames encoder
Flow control handler
Validators
User Handler
Stream N
Implements read/write flow control as usual.
Encodes HTTP/2 frames to Buffer
Intercepts written window_update and priority frames
Validates frame order and content for request/response
stream.write(frame)
stream.write(frame)

NOTABLE API CHANGES

NOTABLE API CHANGES

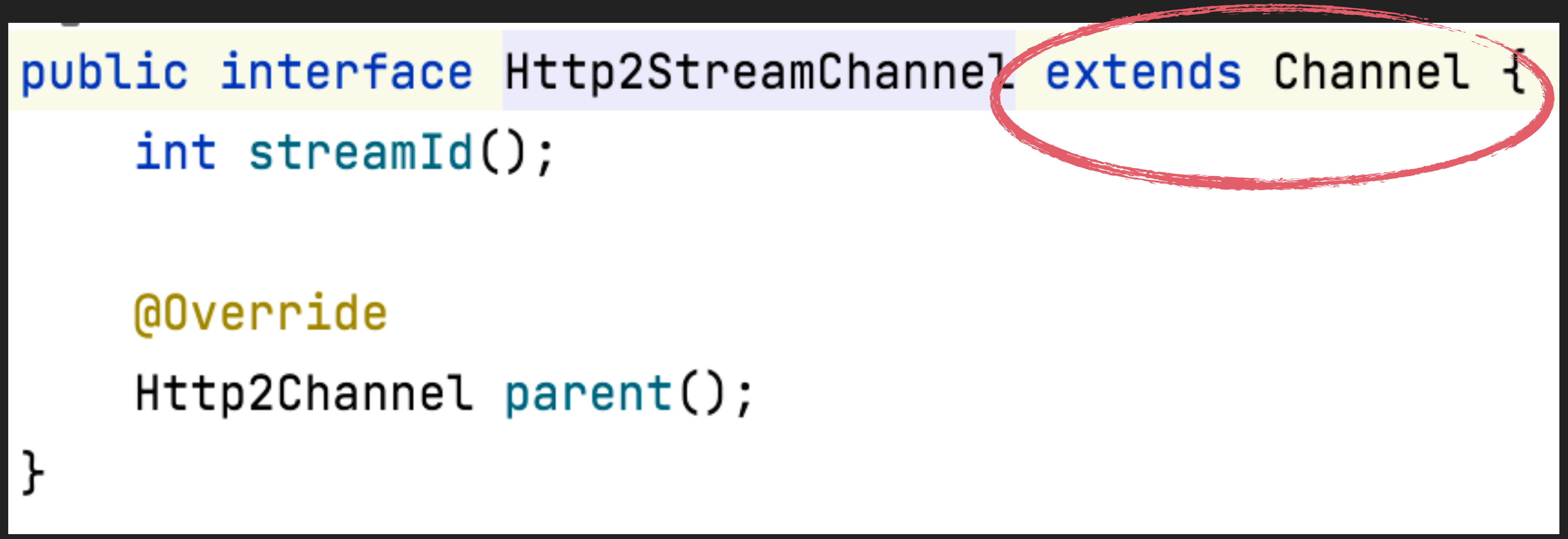
Http2Channel



```java
public interface Http2Channel extends Channel {
    /**
     * Creates a new {@link Http2StreamChannel}. <p>
     * See {@link #newStreamBootstrap()} to use a {@link Http2StreamChannelBootstrap} for creating a
     * {@link Http2StreamChannel}.
     *
     * @param handler to add to the created {@link Http2StreamChannel}.
     * @return the {@link Future} that will be notified once the channel was opened successfully or it failed.
     */
    Future<Http2StreamChannel> createStream(ChannelHandler handler);

    /**
     * Creates a new {@link Http2StreamChannelBootstrap} instance.
     *
     * @return {@link Http2StreamChannelBootstrap}
     */
    default Http2StreamChannelBootstrap newStreamBootstrap() { return new DefaultHttp2StreamChannelBootstrap( parent: this); }
}
```

NOTABLE API CHANGES

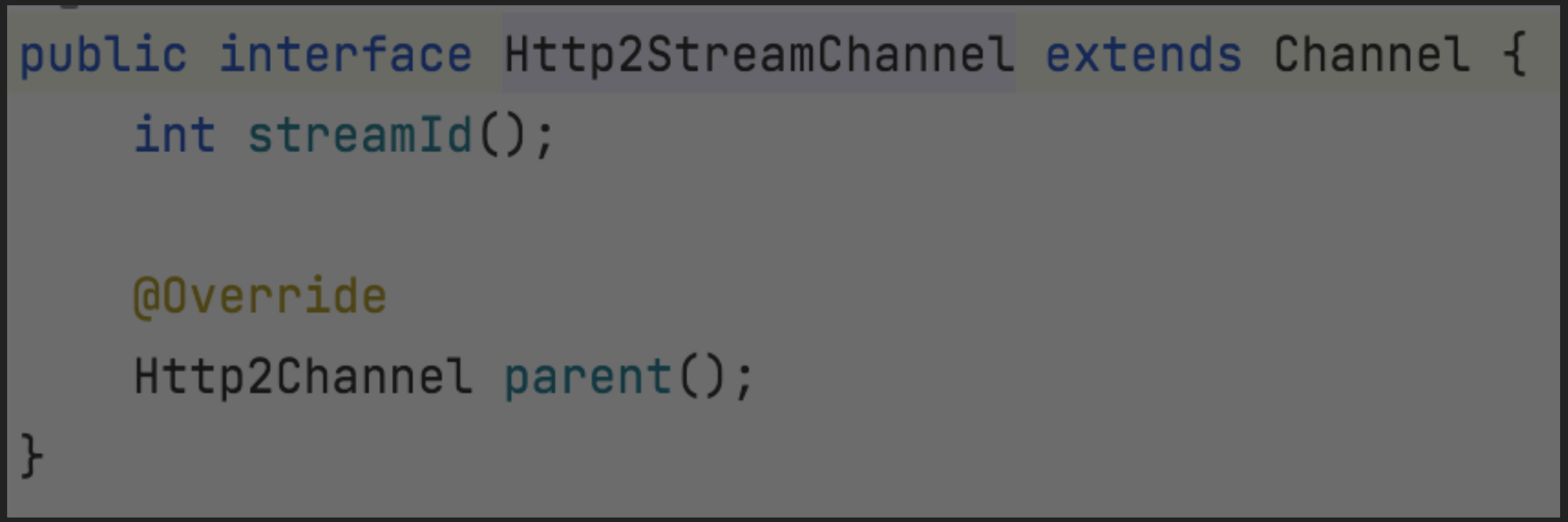
Http2StreamChannel



```
public interface Http2StreamChannel extends Channel {
    int streamId();

    @Override
    Http2Channel parent();
}
```

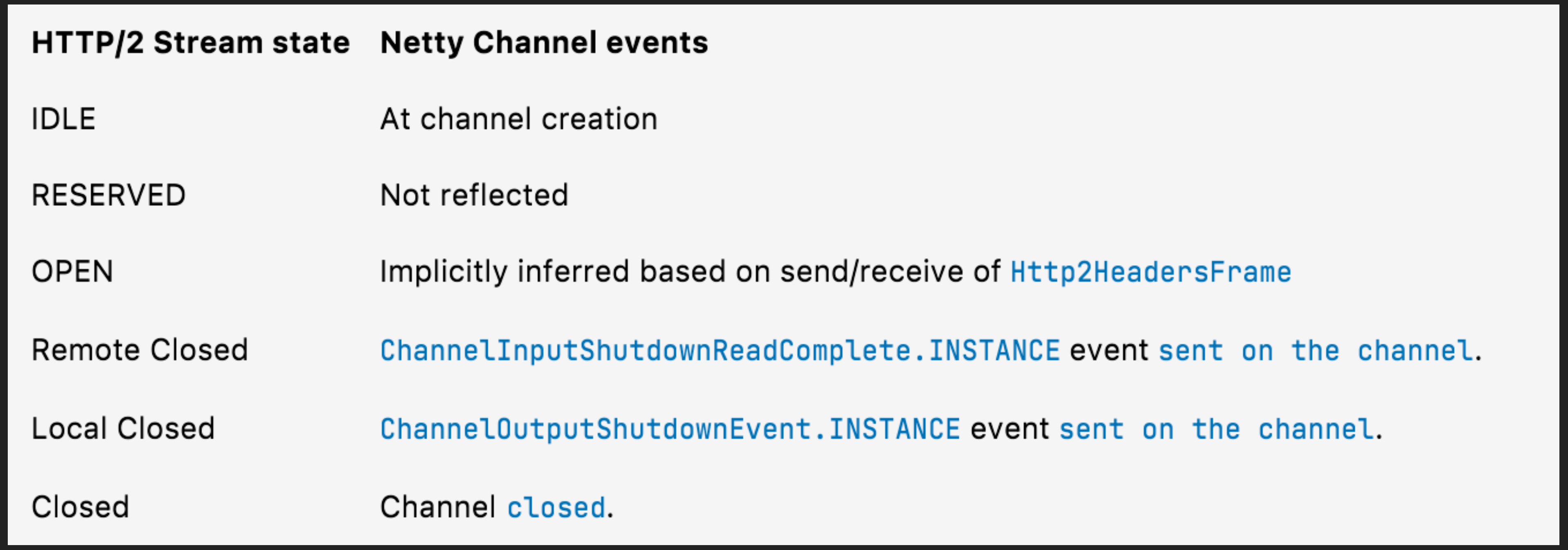
NOTABLE API CHANGES

Http2StreamChannel



```
public interface Http2StreamChannel extends Channel {
    int streamId();

    @Override
    Http2Channel parent();
}
```

HTTP2 Stream states mapping

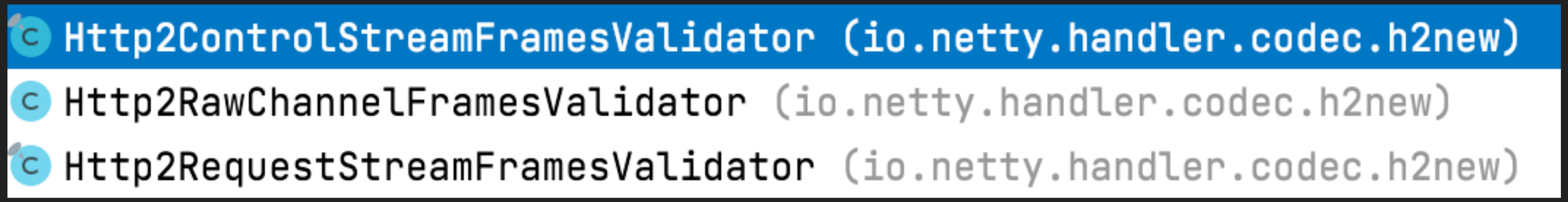


HTTP/2 Stream state	Netty Channel events
IDLE	At channel creation
RESERVED	Not reflected
OPEN	Implicitly inferred based on send/receive of `Http2HeadersFrame`
Remote Closed	`ChannelInputShutdownReadComplete.INSTANCE` event `sent on the channel`.
Local Closed	`ChannelOutputShutdownEvent.INSTANCE` event `sent on the channel`.
Closed	Channel `closed`.

NOTABLE API CHANGES

Validators

All HTTP/2 frame level validations encapsulated in specific handlers.

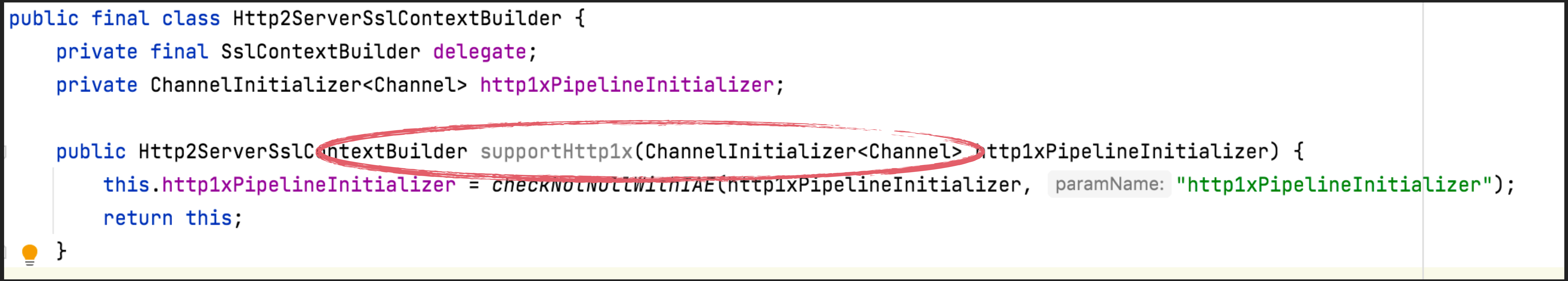


```
Http2ControlStreamFramesValidator (io.netty.handler.codec.h2new)
Http2RawChannelFramesValidator (io.netty.handler.codec.h2new)
Http2RequestStreamFramesValidator (io.netty.handler.codec.h2new)
```

NOTABLE API CHANGES

Custom HTTP/2 SSL context

Hides ALPN upgrade configuration



```
public final class Http2ServerSslContextBuilder {
    private final SslContextBuilder delegate;
    private ChannelInitializer<Channel> http1xPipelineInitializer;

    public Http2ServerSslContextBuilder supportHttp1x(ChannelInitializer<Channel> http1xPipelineInitializer) {
        this.http1xPipelineInitializer = checkNotNullWithIAE(http1xPipelineInitializer, paramName: "http1xPipelineInitializer");
        return this;
    }
```

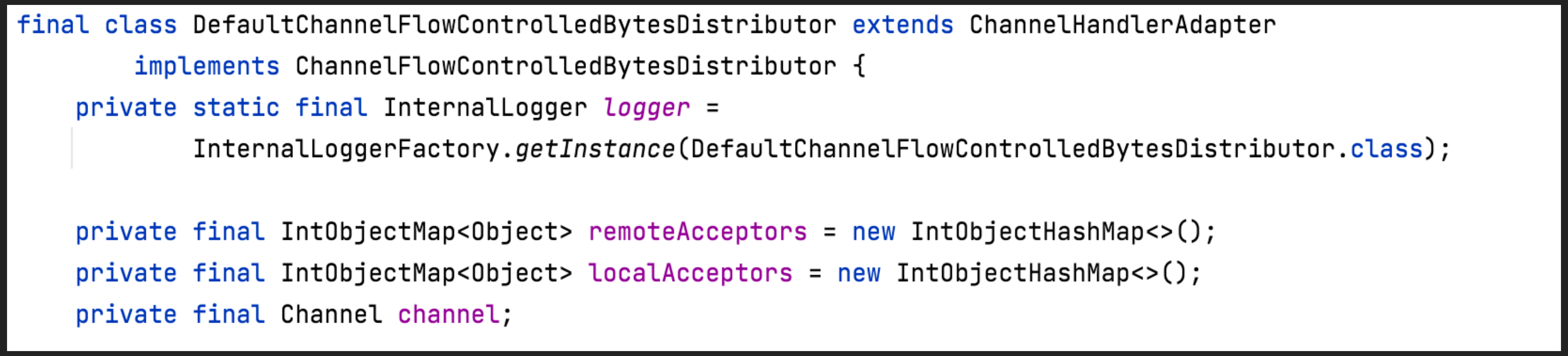
NOTABLE API CHANGES

Flow control

- Reduced public APIs.
  - Plugging in new user-defined algorithms will require more work.

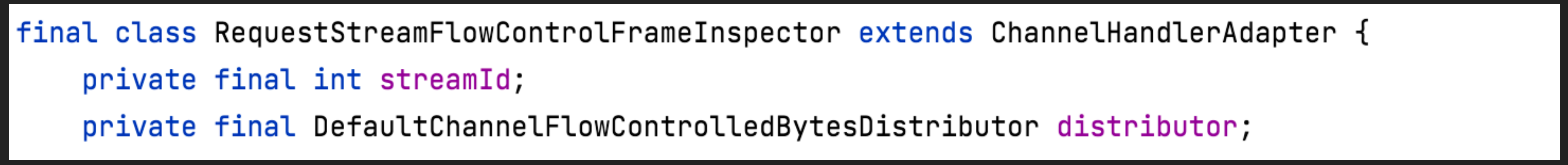
NOTABLE API CHANGES

Flow control



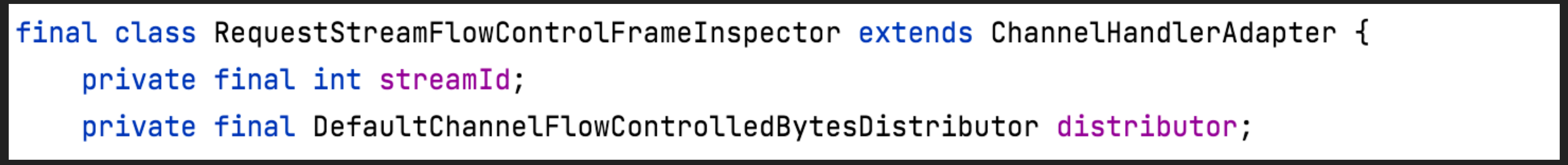
```java
final class DefaultChannelFlowControlledBytesDistributor extends ChannelHandlerAdapter
        implements ChannelFlowControlledBytesDistributor {
    private static final InternalLogger logger =
            InternalLoggerFactory.getInstance(DefaultChannelFlowControlledBytesDistributor.class);

    private final IntObjectMap<Object> remoteAcceptors = new IntObjectHashMap<>();
    private final IntObjectMap<Object> localAcceptors = new IntObjectHashMap<>();
    private final Channel channel;
```

Parent channel handler



```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

. . .

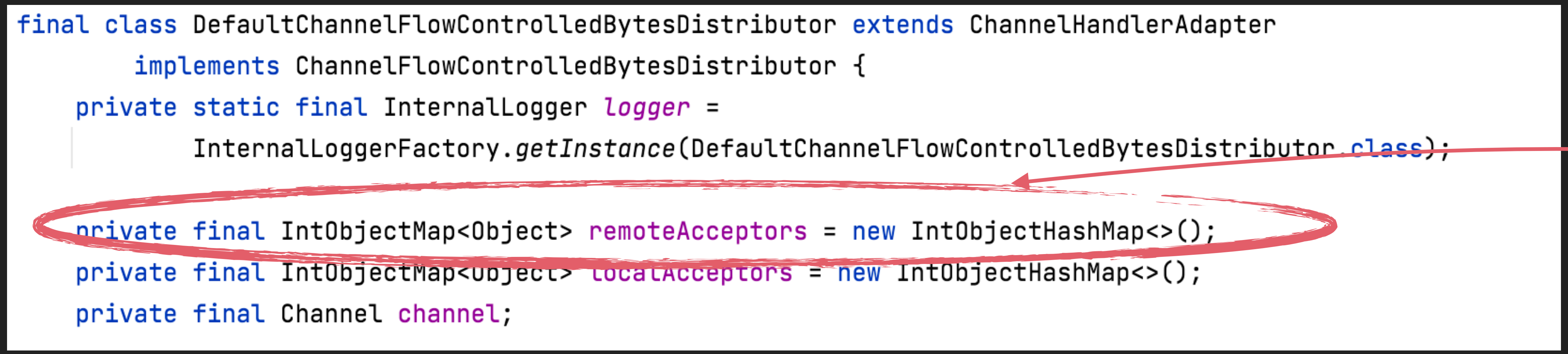


```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

Per stream handlers

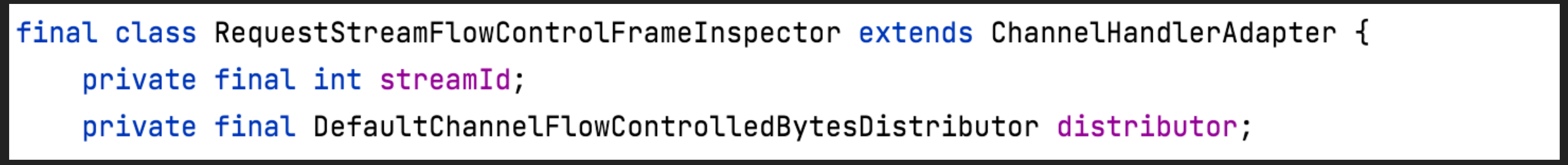
NOTABLE API CHANGES

Flow control



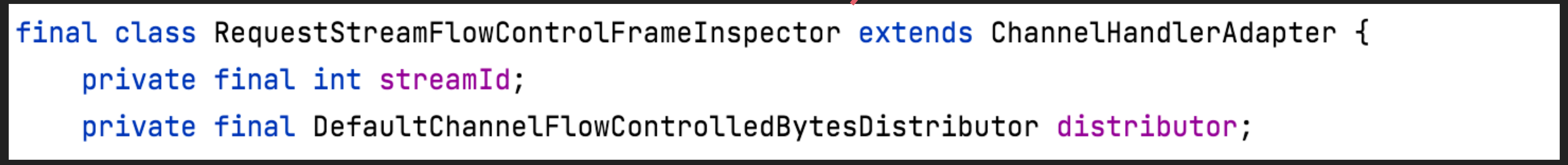
```java
final class DefaultChannelFlowControlledBytesDistributor extends ChannelHandlerAdapter
        implements ChannelFlowControlledBytesDistributor {
    private static final InternalLogger logger =
            InternalLoggerFactory.getInstance(DefaultChannelFlowControlledBytesDistributor.class);

    private final IntObjectMap<Object> remoteAcceptors = new IntObjectHashMap<>();
    private final IntObjectMap<Object> localAcceptors = new IntObjectHashMap<>();
    private final Channel channel;
```

Registers for write flow control credit distribution



```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

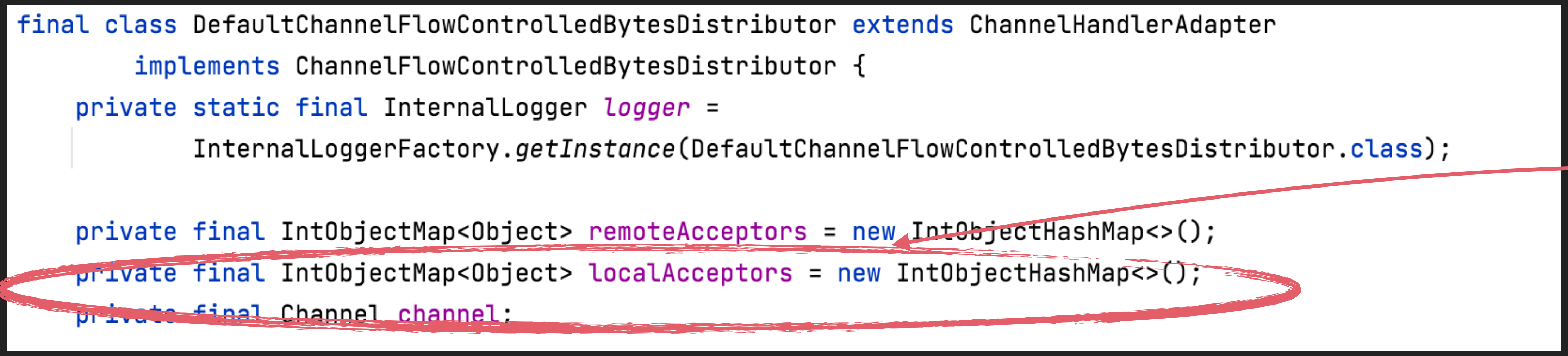
. . .



```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

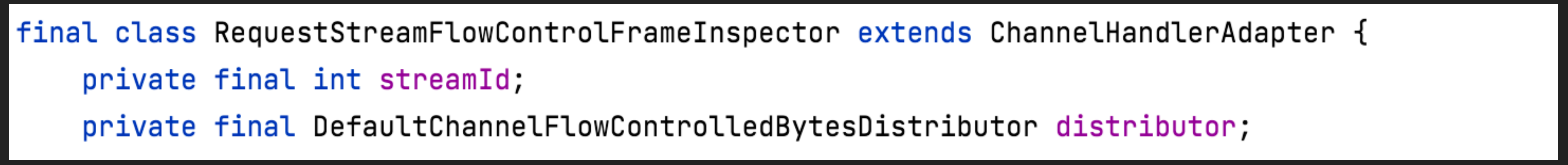
NOTABLE API CHANGES

Flow control



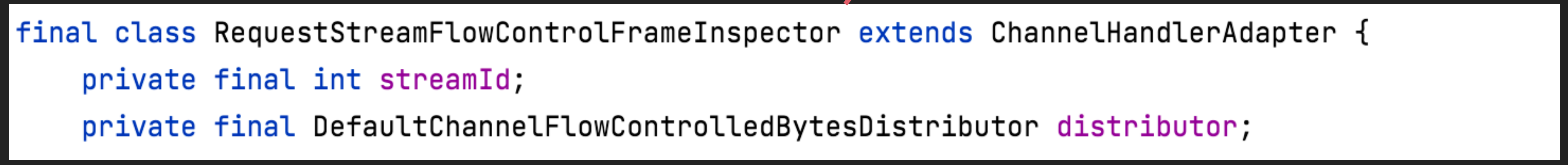
```java
final class DefaultChannelFlowControlledBytesDistributor extends ChannelHandlerAdapter
        implements ChannelFlowControlledBytesDistributor {
    private static final InternalLogger logger =
            InternalLoggerFactory.getInstance(DefaultChannelFlowControlledBytesDistributor.class);

    private final IntObjectMap<Object> remoteAcceptors = new IntObjectHashMap<>();
    private final IntObjectMap<Object> localAcceptors = new IntObjectHashMap<>();
    private final Channel channel;
```

Registers for read flow control credit distribution



```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

. . .



```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

NOTABLE API CHANGES

Flow control

```java
final class DefaultChannelFlowControlledBytesDistributor extends ChannelHandlerAdapter
        implements ChannelFlowControlledBytesDistributor {
    private static final InternalLogger logger =
            InternalLoggerFactory.getInstance(DefaultChannelFlowControlledBytesDistributor.class);

    private final IntObjectMap<Object> remoteAcceptors = new IntObjectHashMap<>();
    private final IntObjectMap<Object> localAcceptors = new IntObjectHashMap<>();
    private final Channel channel;
```

Callbacks when bytes are read/written

```java
distributor.bytesWritten(streamId, bytes);
```

```java
distributor.bytesRead(streamId, ((Http2DataFrame) msg).initialFlowControlledBytes());
```

```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

. . .

```java
final class RequestStreamFlowControlFrameInspector extends ChannelHandlerAdapter {
    private final int streamId;
    private final DefaultChannelFlowControlledBytesDistributor distributor;
```

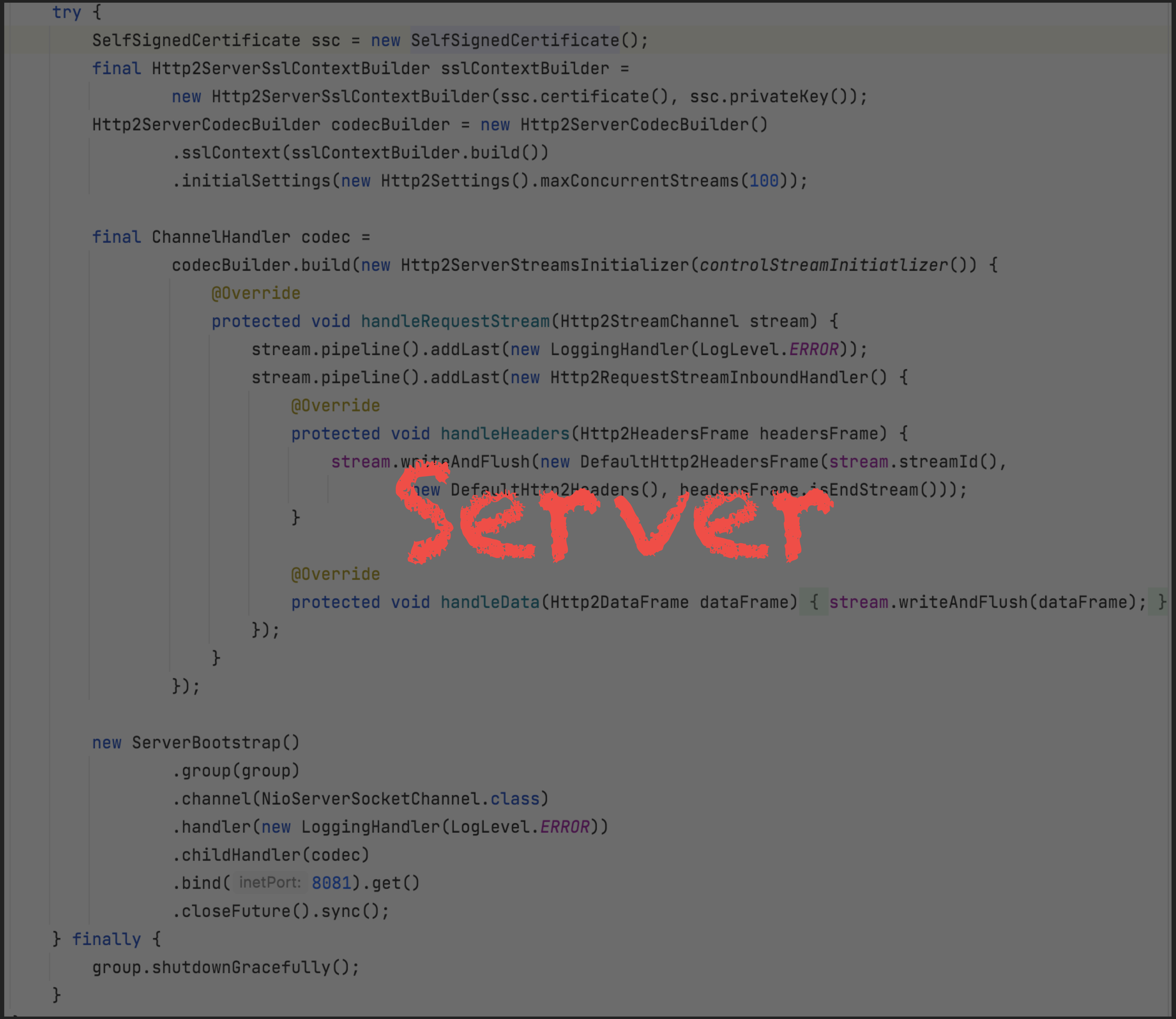
NOTABLE API CHANGES

Experimental raw channel API

- No child channels
  - State per stream can be maintained by users.
  - More control for users at the cost of more plumbing.

EXAMPLES

EXAMPLES

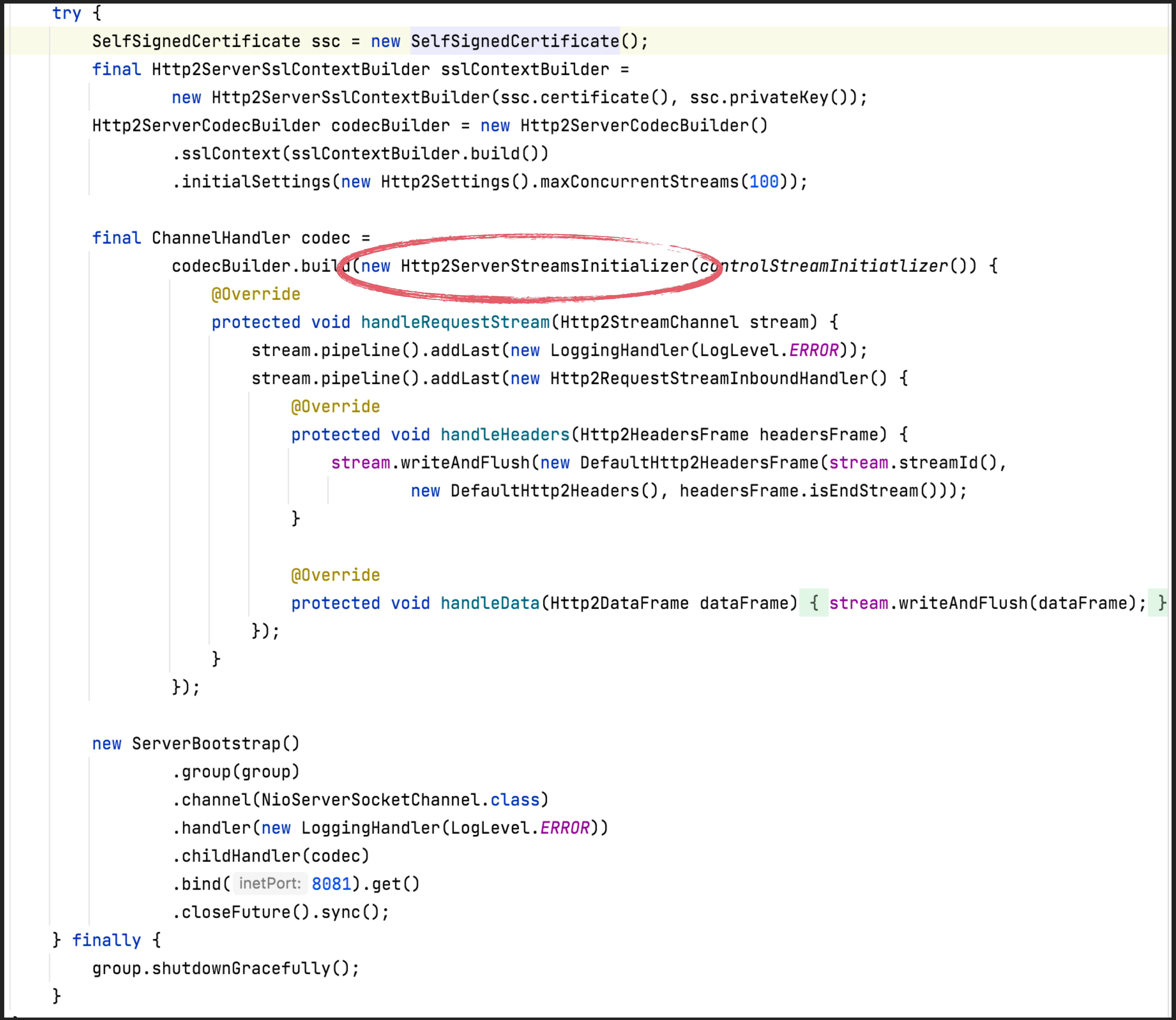


```java
try {
    SelfSignedCertificate ssc = new SelfSignedCertificate();
    final Http2ServerSslContextBuilder sslContextBuilder =
            new Http2ServerSslContextBuilder(ssc.certificate(), ssc.privateKey());
    Http2ServerCodecBuilder codecBuilder = new Http2ServerCodecBuilder()
            .sslContext(sslContextBuilder.build())
            .initialSettings(new Http2Settings().maxConcurrentStreams(100));

    final ChannelHandler codec =
            codecBuilder.build(new Http2ServerStreamsInitializer(controlStreamInitiatlizer()) {
                @Override
                protected void handleRequestStream(Http2StreamChannel stream) {
                    stream.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    stream.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                        @Override
                        protected void handleHeaders(Http2HeadersFrame headersFrame) {
                            stream.writeAndFlush(new DefaultHttp2HeadersFrame(stream.streamId(),
                                    new DefaultHttp2Headers(), headersFrame.isEndStream()));
                        }

                        @Override
                        protected void handleData(Http2DataFrame dataFrame) { stream.writeAndFlush(dataFrame); }
                    });
                }
            });

    new ServerBootstrap()
            .group(group)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.ERROR))
            .childHandler(codec)
            .bind( inetPort: 8081).get()
            .closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

HTTP/2 API CHANGES

EXAMPLES

Server

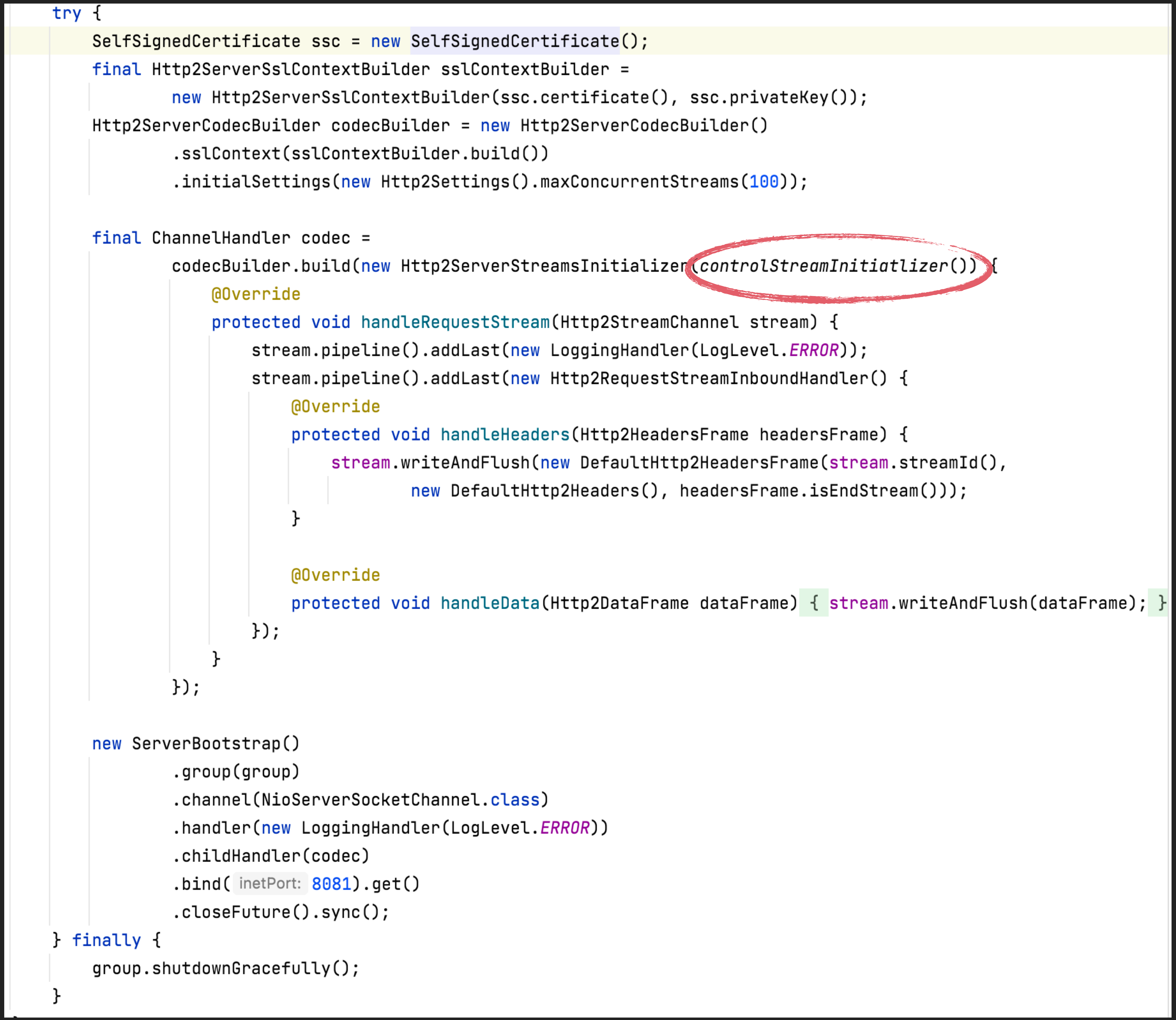


```
try {
    SelfSignedCertificate ssc = new SelfSignedCertificate();
    final Http2ServerSslContextBuilder sslContextBuilder =
            new Http2ServerSslContextBuilder(ssc.certificate(), ssc.privateKey());
    Http2ServerCodecBuilder codecBuilder = new Http2ServerCodecBuilder()
            .sslContext(sslContextBuilder.build())
            .initialSettings(new Http2Settings().maxConcurrentStreams(100));

    final ChannelHandler codec =
            codecBuilder.build(new Http2ServerStreamsInitializer(controlStreamInitiatlizer()) {
                @Override
                protected void handleRequestStream(Http2StreamChannel stream) {
                    stream.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    stream.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                        @Override
                        protected void handleHeaders(Http2HeadersFrame headersFrame) {
                            stream.writeAndFlush(new DefaultHttp2HeadersFrame(stream.streamId(),
                                    new DefaultHttp2Headers(), headersFrame.isEndStream()));
                        }

                        @Override
                        protected void handleData(Http2DataFrame dataFrame) { stream.writeAndFlush(dataFrame); }
                    });
                }
            });

    new ServerBootstrap()
            .group(group)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.ERROR))
            .childHandler(codec)
            .bind( inetPort: 8081).get()
            .closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

Out of the box initializer invoked for each connection

EXAMPLES

Server

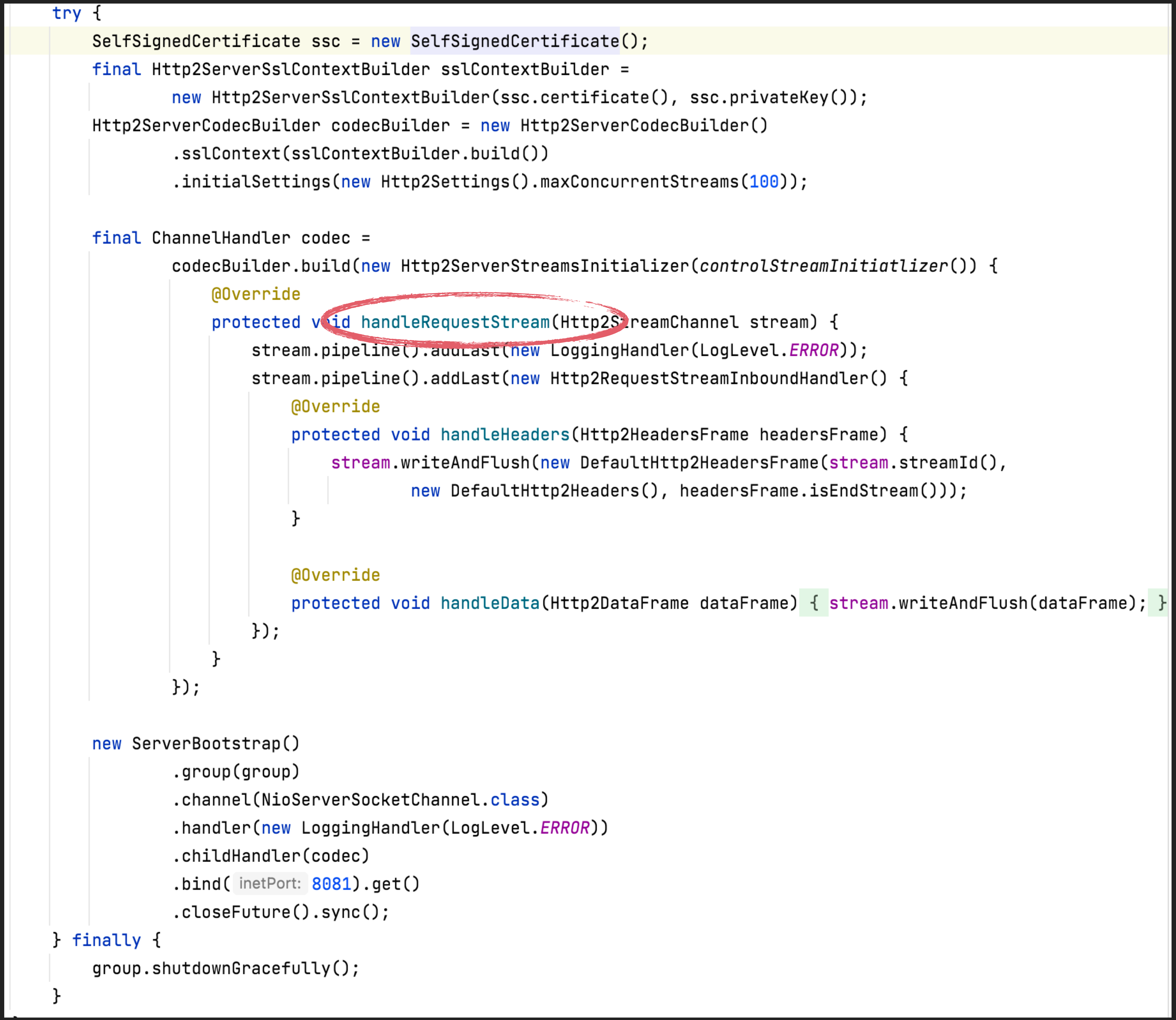


```
try {
    SelfSignedCertificate ssc = new SelfSignedCertificate();
    final Http2ServerSslContextBuilder sslContextBuilder =
            new Http2ServerSslContextBuilder(ssc.certificate(), ssc.privateKey());
    Http2ServerCodecBuilder codecBuilder = new Http2ServerCodecBuilder()
            .sslContext(sslContextBuilder.build())
            .initialSettings(new Http2Settings().maxConcurrentStreams(100));

    final ChannelHandler codec =
            codecBuilder.build(new Http2ServerStreamsInitializer(controlStreamInitiatlizer()) {
                @Override
                protected void handleRequestStream(Http2StreamChannel stream) {
                    stream.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    stream.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                        @Override
                        protected void handleHeaders(Http2HeadersFrame headersFrame) {
                            stream.writeAndFlush(new DefaultHttp2HeadersFrame(stream.streamId(),
                                    new DefaultHttp2Headers(), headersFrame.isEndStream()));
                        }

                        @Override
                        protected void handleData(Http2DataFrame dataFrame) { stream.writeAndFlush(dataFrame); }
                    });
                }
            });

    new ServerBootstrap()
            .group(group)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.ERROR))
            .childHandler(codec)
            .bind( inetPort: 8081).get()
            .closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

Optionally configure control stream channel

EXAMPLES

Server



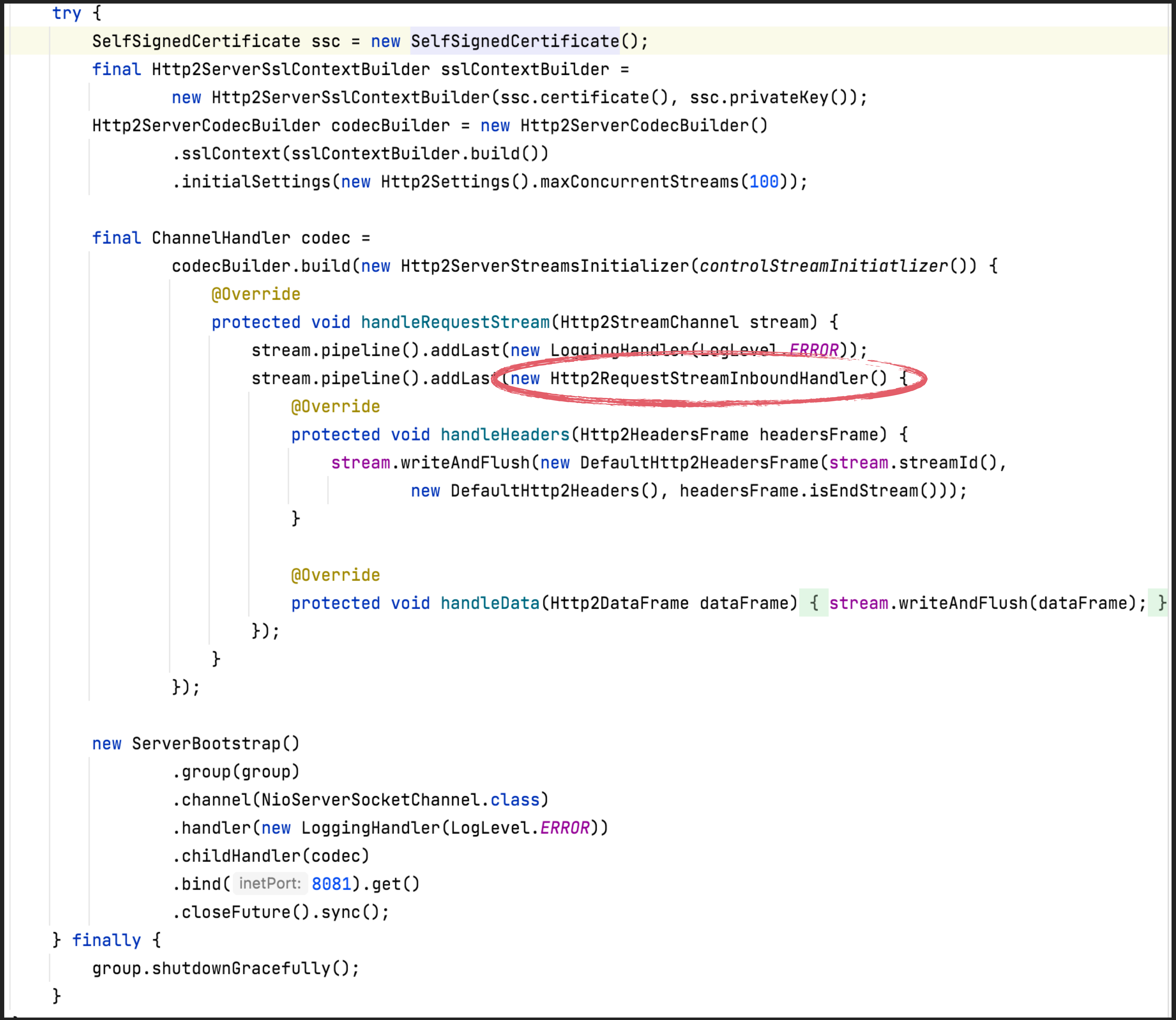
```java
try {
    SelfSignedCertificate ssc = new SelfSignedCertificate();
    final Http2ServerSslContextBuilder sslContextBuilder =
            new Http2ServerSslContextBuilder(ssc.certificate(), ssc.privateKey());
    Http2ServerCodecBuilder codecBuilder = new Http2ServerCodecBuilder()
            .sslContext(sslContextBuilder.build())
            .initialSettings(new Http2Settings().maxConcurrentStreams(100));

    final ChannelHandler codec =
            codecBuilder.build(new Http2ServerStreamsInitializer(controlStreamInitiatlizer()) {
                @Override
                protected void handleRequestStream(Http2StreamChannel stream) {
                    stream.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    stream.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                        @Override
                        protected void handleHeaders(Http2HeadersFrame headersFrame) {
                            stream.writeAndFlush(new DefaultHttp2HeadersFrame(stream.streamId(),
                                    new DefaultHttp2Headers(), headersFrame.isEndStream()));
                        }

                        @Override
                        protected void handleData(Http2DataFrame dataFrame) { stream.writeAndFlush(dataFrame); }
                    });
                }
            });

    new ServerBootstrap()
            .group(group)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.ERROR))
            .childHandler(codec)
            .bind( inetPort: 8081).get()
            .closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

Configure each accepted request stream channel

HTTP/2 API CHANGES

EXAMPLES

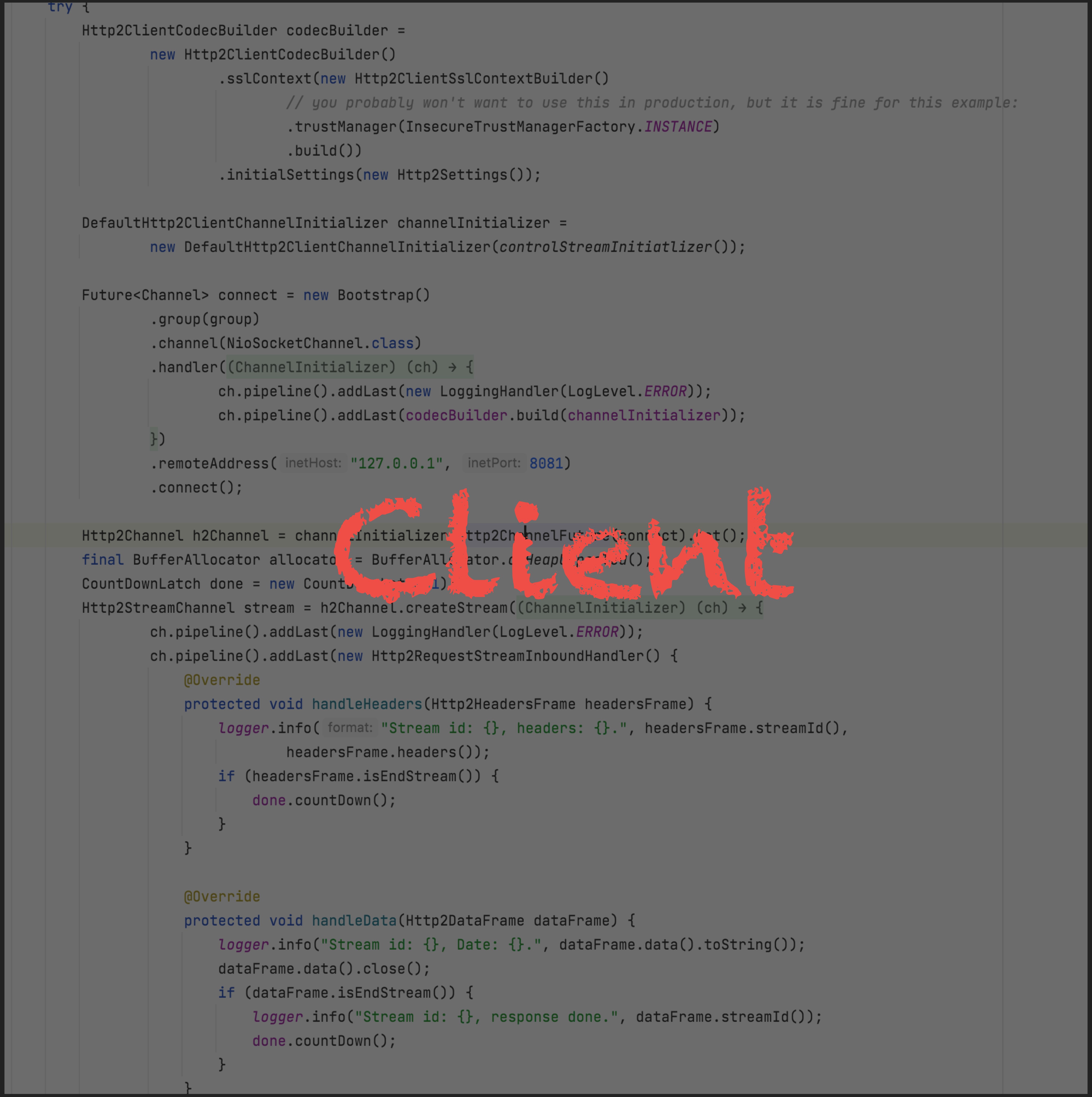
Server



```java
try {
    SelfSignedCertificate ssc = new SelfSignedCertificate();
    final Http2ServerSslContextBuilder sslContextBuilder =
            new Http2ServerSslContextBuilder(ssc.certificate(), ssc.privateKey());
    Http2ServerCodecBuilder codecBuilder = new Http2ServerCodecBuilder()
            .sslContext(sslContextBuilder.build())
            .initialSettings(new Http2Settings().maxConcurrentStreams(100));

    final ChannelHandler codec =
            codecBuilder.build(new Http2ServerStreamsInitializer(controlStreamInitiatlizer()) {
                @Override
                protected void handleRequestStream(Http2StreamChannel stream) {
                    stream.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    stream.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                        @Override
                        protected void handleHeaders(Http2HeadersFrame headersFrame) {
                            stream.writeAndFlush(new DefaultHttp2HeadersFrame(stream.streamId(),
                                    new DefaultHttp2Headers(), headersFrame.isEndStream()));
                        }

                        @Override
                        protected void handleData(Http2DataFrame dataFrame) { stream.writeAndFlush(dataFrame); }
                    });
                }
            });

    new ServerBootstrap()
            .group(group)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.ERROR))
            .childHandler(codec)
            .bind( inetPort: 8081).get()
            .closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

Out of the box request handler

EXAMPLES

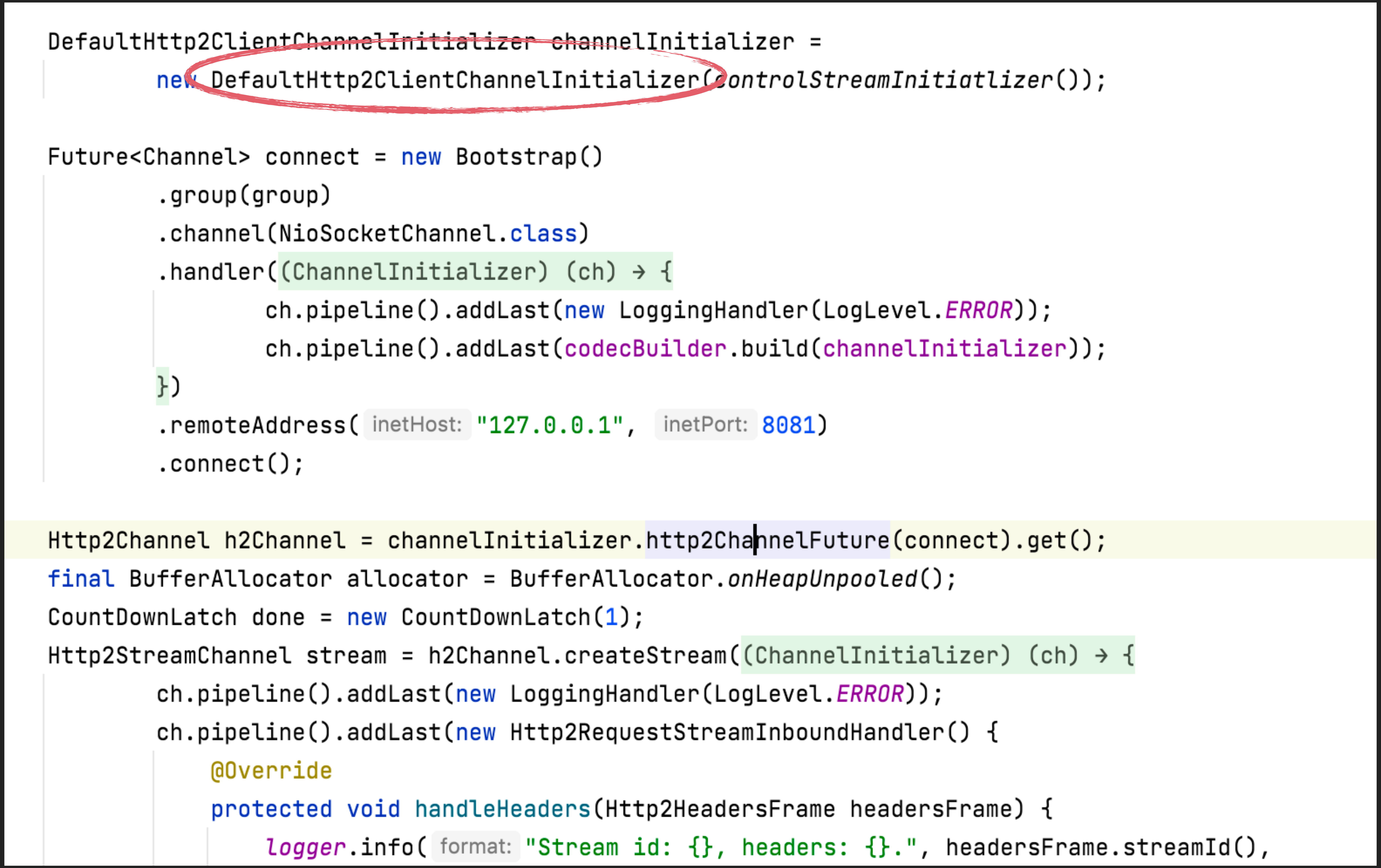


```java
try {
    Http2ClientCodecBuilder codecBuilder =
            new Http2ClientCodecBuilder()
                    .sslContext(new Http2ClientSslContextBuilder()
                            // you probably won't want to use this in production, but it is fine for this example:
                            .trustManager(InsecureTrustManagerFactory.INSTANCE)
                            .build())
                    .initialSettings(new Http2Settings());

    DefaultHttp2ClientChannelInitializer channelInitializer =
            new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

    Future<Channel> connect = new Bootstrap()
            .group(group)
            .channel(NioSocketChannel.class)
            .handler((ChannelInitializer) (ch) → {
                    ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                    ch.pipeline().addLast(codecBuilder.build(channelInitializer));
            })
            .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
            .connect();

    Http2Channel h2Channel = chann[illegible]Initializer[illegible]ttp2Ch[illegible]nelFu[illegible]conn[illegible]ct).[illegible]t();
    final BufferAllocator allocato[illegible] = BufferAll[illegible]ator.[illegible]Heap[illegible]();
    CountDownLatch done = new Count[illegible]1);
    Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
            ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
            ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
                @Override
                protected void handleHeaders(Http2HeadersFrame headersFrame) {
                    logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
                            headersFrame.headers());
                    if (headersFrame.isEndStream()) {
                        done.countDown();
                    }
                }

                @Override
                protected void handleData(Http2DataFrame dataFrame) {
                    logger.info("Stream id: {}, Date: {}.", dataFrame.data().toString());
                    dataFrame.data().close();
                    if (dataFrame.isEndStream()) {
                        logger.info("Stream id: {}, response done.", dataFrame.streamId());
                        done.countDown();
                    }
                }
```

HTTP/2 API CHANGES

EXAMPLES

Client



```
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

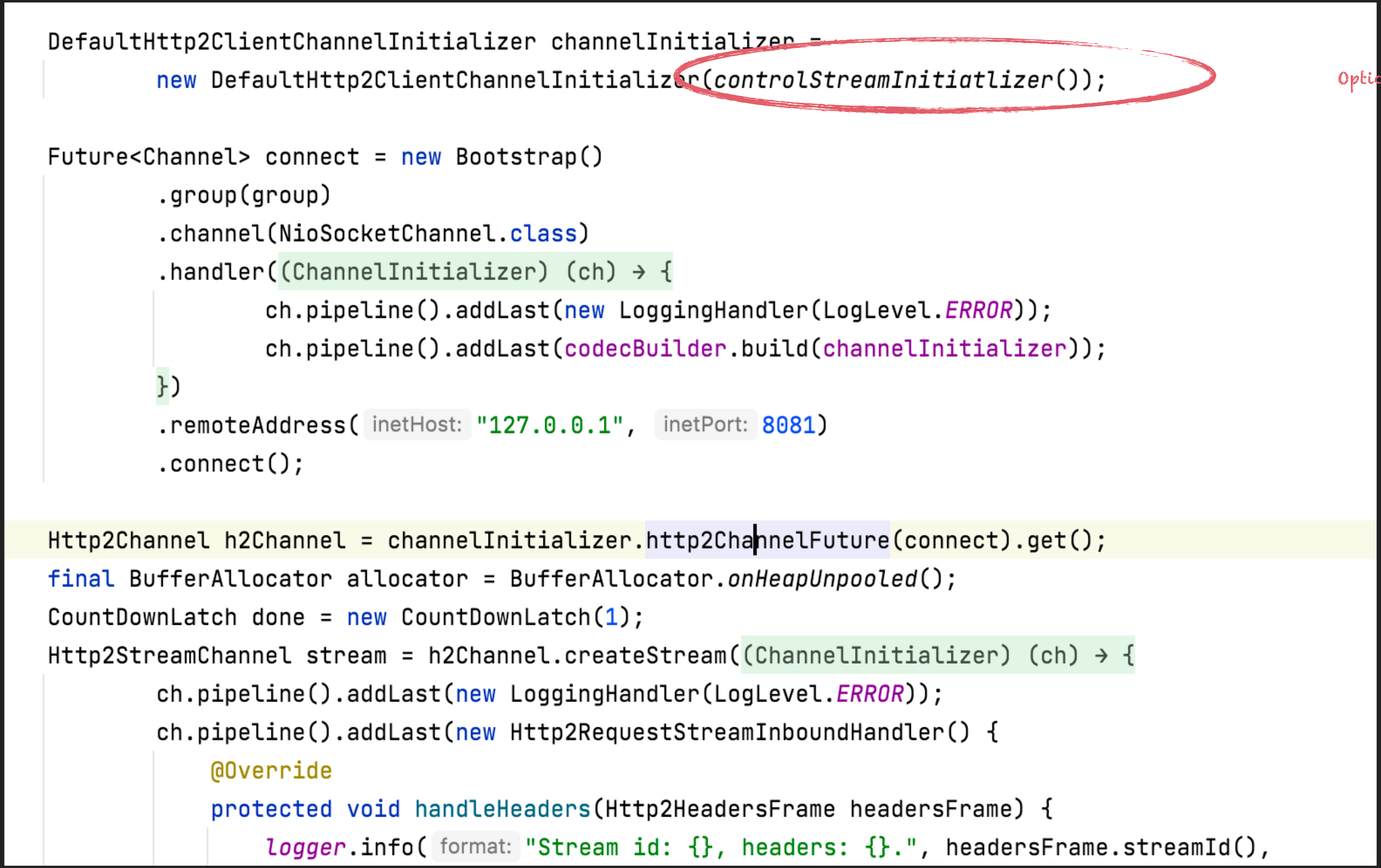
Out of the box initializer

created for each connection

HTTP/2 API CHANGES

EXAMPLES

Client

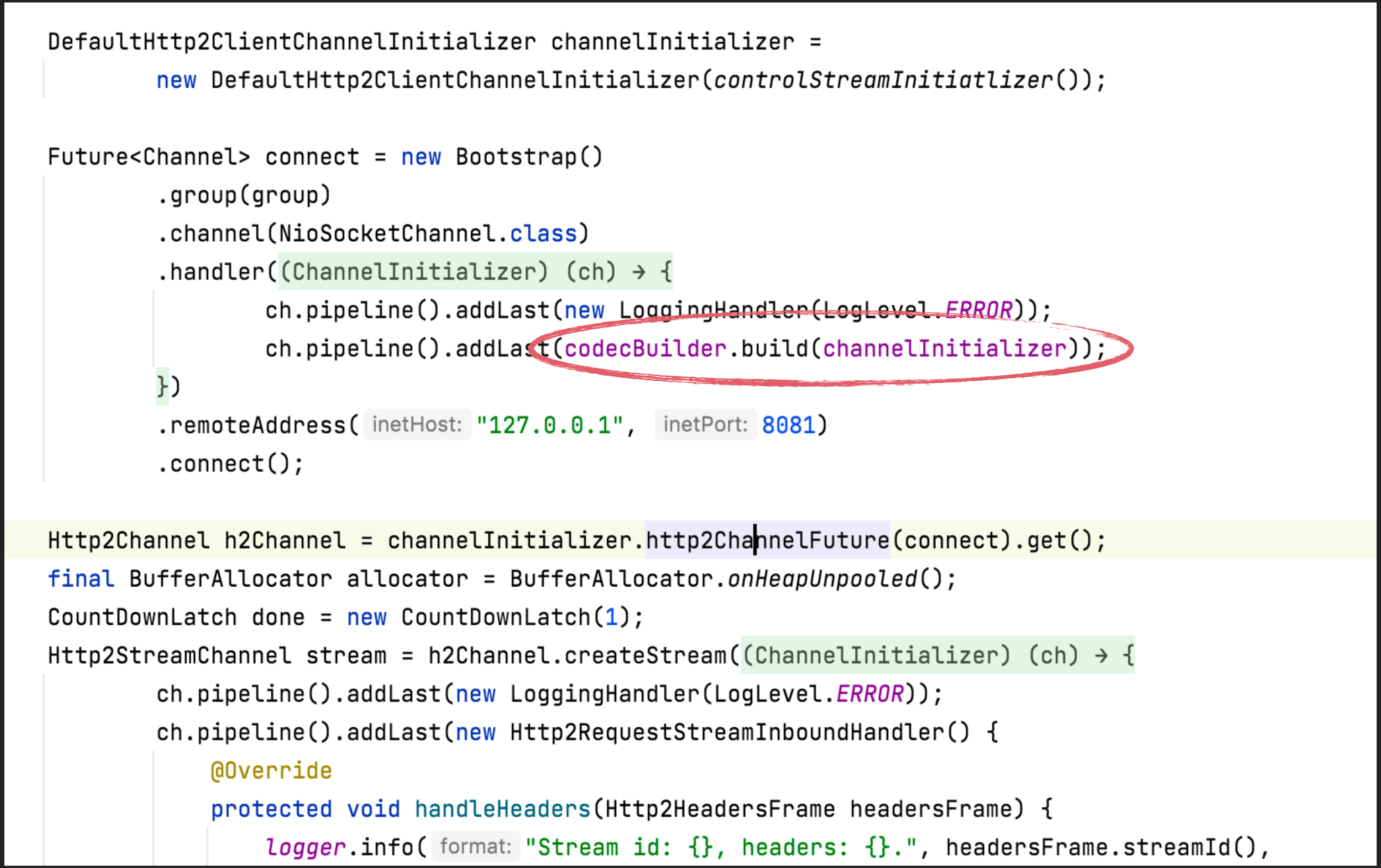


```
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

Optionally configure control stream channel

EXAMPLES

Client



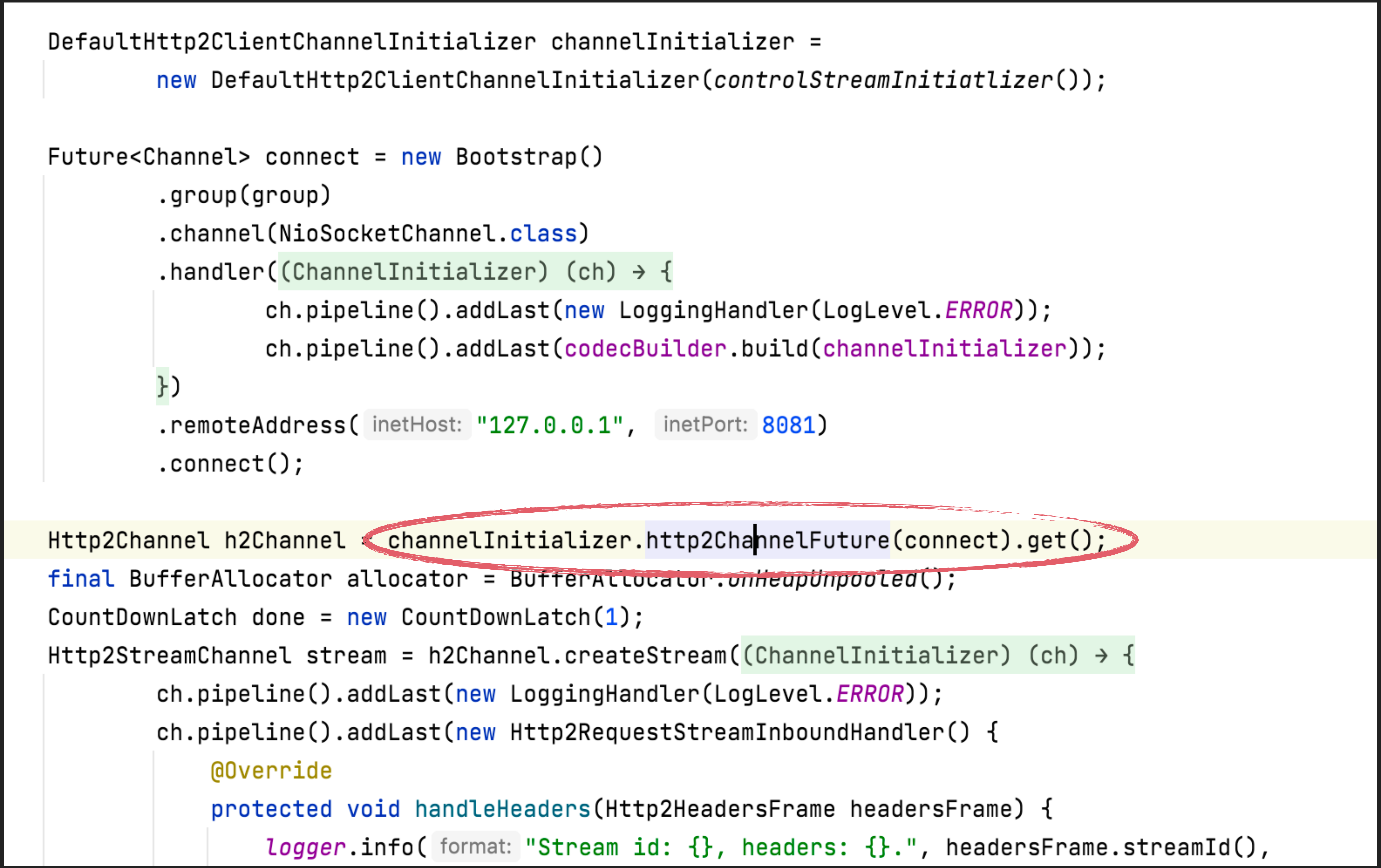
```
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

Supply the initializer to the codec

HTTP/2 API CHANGES

EXAMPLES

Client



```
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

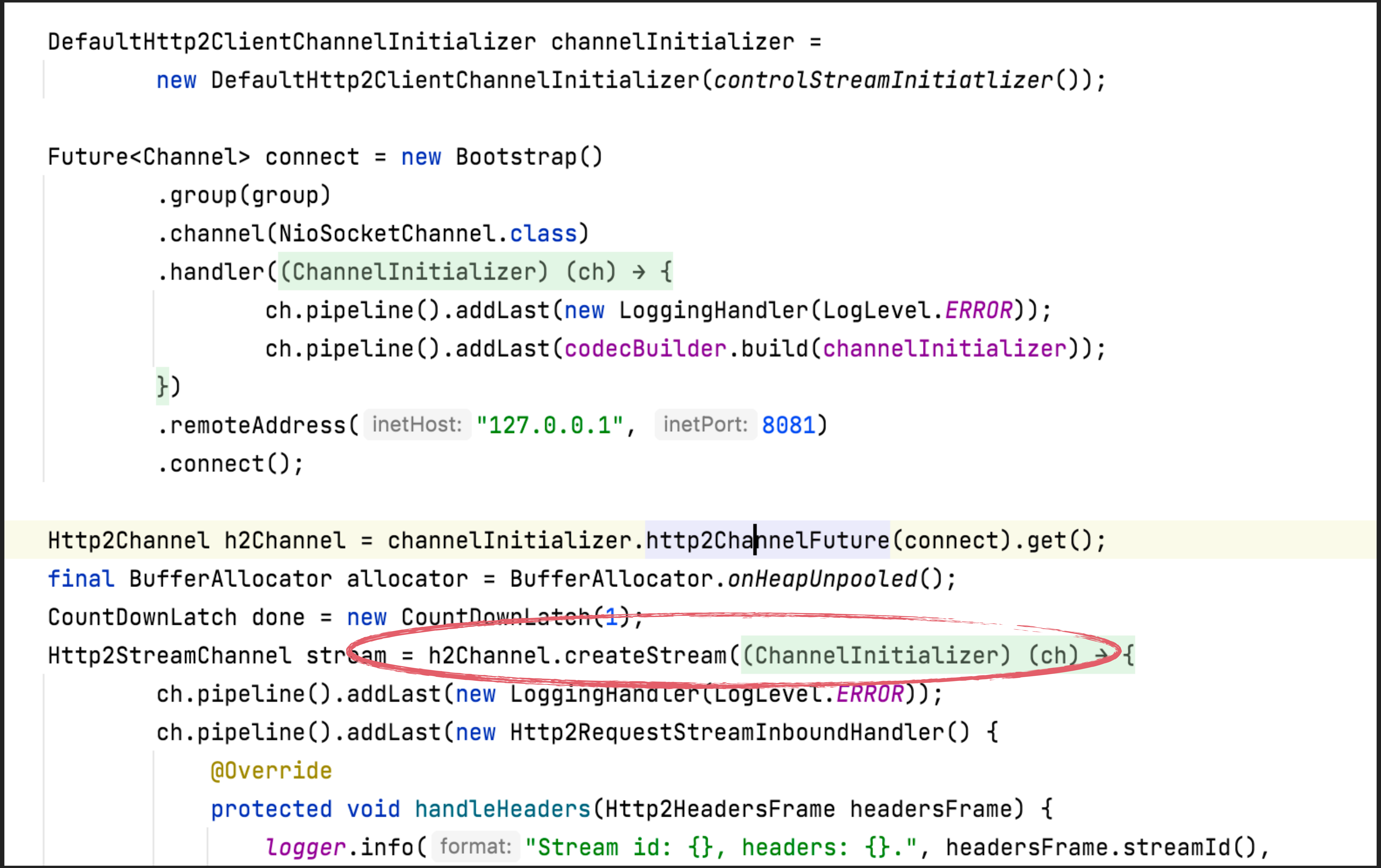
Convert connect future

to Http2Channel future

HTTP/2 API CHANGES

EXAMPLES

Client



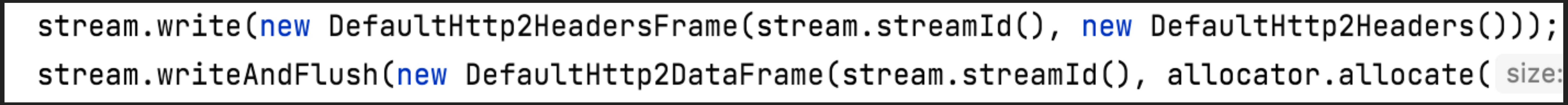
```java
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

Create a stream

for request/response

EXAMPLES

Client

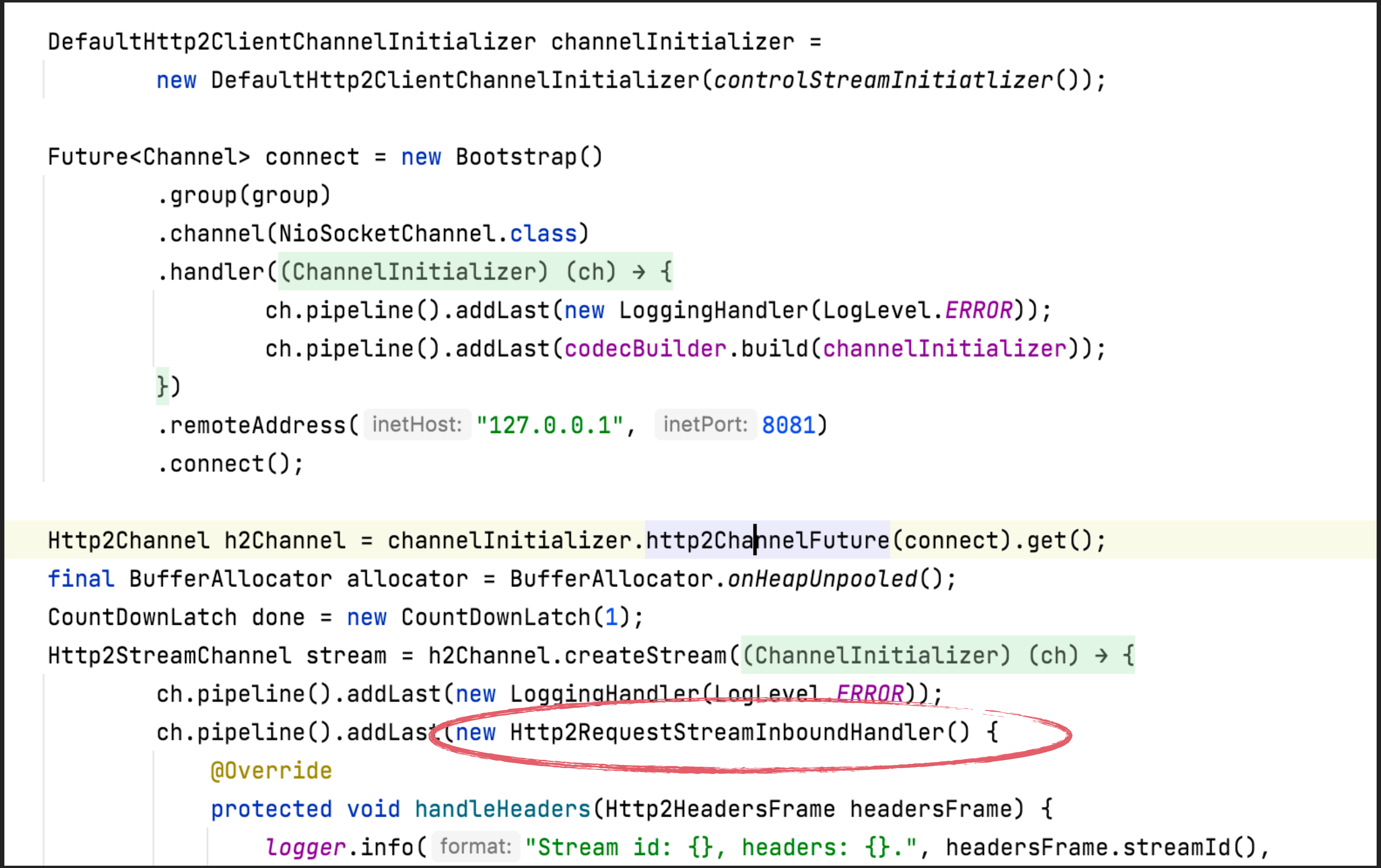


```
stream.write(new DefaultHttp2HeadersFrame(stream.streamId(), new DefaultHttp2Headers()));
stream.writeAndFlush(new DefaultHttp2DataFrame(stream.streamId(), allocator.allocate( size:
```

Write to the stream

EXAMPLES

Client



```
DefaultHttp2ClientChannelInitializer channelInitializer =
        new DefaultHttp2ClientChannelInitializer(controlStreamInitiatlizer());

Future<Channel> connect = new Bootstrap()
        .group(group)
        .channel(NioSocketChannel.class)
        .handler((ChannelInitializer) (ch) → {
                ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
                ch.pipeline().addLast(codecBuilder.build(channelInitializer));
        })
        .remoteAddress( inetHost: "127.0.0.1",  inetPort: 8081)
        .connect();

Http2Channel h2Channel = channelInitializer.http2ChannelFuture(connect).get();
final BufferAllocator allocator = BufferAllocator.onHeapUnpooled();
CountDownLatch done = new CountDownLatch(1);
Http2StreamChannel stream = h2Channel.createStream((ChannelInitializer) (ch) → {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.ERROR));
        ch.pipeline().addLast(new Http2RequestStreamInboundHandler() {
            @Override
            protected void handleHeaders(Http2HeadersFrame headersFrame) {
                logger.info( format: "Stream id: {}, headers: {}.", headersFrame.streamId(),
```

Read response using

Out of the box handler

THANK YOU

QUESTIONS?