Transferable Streams Stage 1 Design Document

Status: implemented behind a flag

Objective

Permit transferring a stream to other realms using postMessage(). In stage 1, chunks enqueued in such a stream will be cloned, ie. all data will be copied. Support for transferring chunks (zero copy) is stage 2, which is not covered by this document.

This is an example of JavaScript which will work once this is implemented:

In the main page:

```
const rs = new ReadableStream({
  start(controller) {
    controller.enqueue('hello');
  }
});
const w = new Worker('worker.js');
w.postMessage(rs, [rs]);
```

In worker.js:

```
onmessage = async (evt) => {
  const rs = evt.data;
  const reader = rs.getReader();
  const {value, done} = await reader.read();
  console.log(value); // logs 'hello'.
};
```

Note that `w.postMessage(rs)` would not work. Streams can only be transferred, not cloned. Attempts to clone a stream will throw a DataCloneError.

Background

The streams APIs provide ubiquitous, interoperable primitives for creating, composing, and consuming streams of data. A natural thing to want to do with a stream is pass it off to another thread. This would provide a fluent primitive for offloading work onto another thread.

This work will permit streams to be transferred between workers, frames and anywhere else that postMessage() can be used. Chunks can be anything which is cloneable by postMessage().

Design

The original stream is not actually transferred. It is locked, a new stream is created in the target realm, and then chunks are cloned between them. The new stream behaves like the original.

The main primitive used to achieve this is a cross-realm identity transform. This is a similar in concept to an identity transform created by the TransformStream constructor, but internally the two sides are connected by a MessageChannel.

To transfer a ReadableStream, a cross-realm identity transform is created with the *writable* side in the source realm and the *readable* side in the target realm. The original ReadableStream is

piped to the *writable* side of the cross-realm identity transform as if by pipeTo().

To transfer a WritableStream, a cross-realm identity transform is created with the *readable* side in the source realm and the *writable* side in the target realm. The *readable* side of the cross-realm identity transform is piped to the original WritableStream as if by pipeTo().

To transfer a TransformStream, it is treated as a [readable, writable] pair which are transferred as above, then the pair is assembled into a new TransformStream in the target realm.

Detailed Design

Extract Transferables

The first step in performing `postMessage(..., transfer)` is to check the *transfer* list and prepare the objects for serialisation.

Check the transfer list for duplicates: for this we need sets of the ReadableStream, WritableStream and TransformStream objects that we have already seen. For consistency with other transferable types, we will use Vectors in the blink::Transferables object for this.

Extra conditions will be added to SerializedScriptValue::ExtractTransferables() to perform these operations and record them in blink::Transferables.

Serialization

Serialization is done by blink::V8ScriptValueSerializer which uses v8::ValueSerializer. v8::ValueSerializer calls blink::V8ScriptValueSerializer::WriteHostObject (through v8::ValueSerializer::Delegate interface) when it sees an "embedder object", and serializes other objects by itself. The current criteria is, a v8::internal::JSReceiver is an embedder object if

- its *embedder field count* is greater than zero, or
- its *instance type* is JS_SPECIAL_API_OBJECT_TYPE.

We need serialization logic specific to (Readable|Writable)Stream, so we need to have (Readable|Writable)Stream trigger WriteHostObject. The problem is, there is no easy way to do that given (Readable|Writable)Stream is implemented with V8Extra. We have some options described below.

A: Implement (Readable|Writable)Stream as a ScriptWrappable

Stop using V8Extra and implement streams as ScriptWrappables. This doesn't need changes to V8 or blink bindings.

B: Introduce a way to trigger WriteHostObject for a V8Extra object

This needs changes to V8. We will introduce a special operation visible to V8Extra which makes a (Readable|Writable)Stream an embedder object.

C: Introduce a ScriptWrappable field in Stream

Introduce a field implemented as a ScriptWrappable, say *f*, to (Readable|Writable)Stream. A (Readable|Writable) stream is treated as an ordinary object that only has *f* by v8::ValueSerializer. As *f* is an embedder object, WriteHostObject is called for *f*. We can serialize the (Readable|Writable)Stream when we serialize *f*.

This pollutes the global scope (the interface implemented by the ScriptWrappable newly introduced) and (Readable|Writable)Stream (the newly introduced field must be visible to author scripts in order to be visible to v8::ValueSerializer), so this **cannot** be shipped.

D: Introduce a thin ScriptWrappable

Stop exposing ReadableStream constructor from ReadableStream.js. Instead we define ReadableStream.idl and blink::ReadableStream which contains a TraceWrapperV8Reference to actual ReadableStream instance. Each public IDL methods are delegated to the corresponding V8Extra call via ReadableStreamOperations.

The implementation is still done by V8Extra but stream instances are not directly exposed any more. Instead, there is an additional indirection. The indirection will cause performance loss but I think it's negligible. The lifetime semantics doesn't change as long as ReadableStream wrapper is exposed to scripts.

Implementation strategy

While there are also other reasons to stop using V8Extras (build flakiness, performance, stability, IDL) and we want to do that in the future, it takes time (more than 1Q) and we don't want Transferable Streams to be blocked by that. So we think **A** as a long term solution. **C** is the easiest option to bootstrap an implementation. **D** will take significant amounts of work, but is lower risk and quicker than **A**.

In short, we are going to start development with option **C** first, but because the deserialized structure will be incorrect (explained below under Deserialization), it won't be useful to developers. The V8 team have decided that option **B** cannot be implemented without unacceptable performance overhead. So in parallel we will develop option **D**, and once that is ready switch our implementation to use that.

In future when the C++ port is ready we should be able to easily migrate from option **D** to option **A**.

Common logic

Regardless of how WriteHostObject is called, we need to run some code which locks the stream object and associates it with MessagePort that will be sent to the target realm. First stream objects that weren't included in the *transfer* list should be rejected, as it is not possible to clone them. ReadableStream and WritableStream objects are serialized as the index of the message port that will be used to communicate between the two ends. TransformStream objects are serialized as the index of the first of the pair of message ports that are used to construct the readable and writable sides.

Finally, after the object has been walked and converted to bytes, the original streams need to be locked to a MessagePort. We create a `mojo::MessagePipe` and immediately convert one handle to a message port. The other handle is wrapped in a MessagePortChannel and stored in a `stream_channels_` member on the SerializedScriptValue object.

Then, according to the type of the stream:

- ReadableStreamSerialize(*readable, localPort*) will pass *localPort* to CreateCrossRealmTransformWritable, creating *writable*. It will then call ReadableStreamPipeTo(*readable*, *writable*, false, false, false).
- WritableStreamSerialize(*writable, localPort*) will pass *localPort* to CreateCrossRealmTransformReadable, creating *readable*. It will then call ReadableStreamPipeTo(readable, *writable*, false, false, false).
- Serializing a TransformStream is done by calling ReadableStreamSerialize(*transform*.readable, *localPort1*) and then WritableStreamSerialize(*transform*.writable, *localPort2*).

Transferring *stream_channels*

MessagePorts are represented by `MessagePortChannel` objects during transfer. A `stream_channels` field needs to be added to transferable_message.mojom with the same type as

the existing `ports` field. This also means adding the field to blink::TransferableMessage. blink::BlinkTransferableMessage contains the SerializedScriptValue object with the `stream_channels` field via `blink::BlinkCloneableMessage` and so doesn't need a separate field, but it still needs code to extract the field in its StructTrait definitions. The serialisation/deserialisation ParamTraits for TransferableMessage in content also need to be updated with the new fields.

Deserialization

Deserialization is, in principle, not as hard as serialization. But we would need special logic if we serialize with option **C**, because deserializing what **C** serialized a stream *s* leads to an object that contains *s*, not *s* itself.

In principle, we can put some special logic to blink::V8ScriptValueDeserializer which replaces *{s}* with *s* where *s* is a (Readable|Writable)Stream. However, the outer object can contain arbitrary other fields, and be nested in an arbitrarily deep structure which would need to be traversed and altered. There doesn't appear to be a simple way to do this.

Options **A**, **B** and **D** require no special cases and so are preferable in this respect.

Once we have an array of MessagePorts we can call into JavaScript to create the "transferred" streams in the target realm:

- ReadableStreamDeserialize(port) will call CreateCrossRealmTransformReadable(port).
- WritableStreamDeserialize(port) will call CreateCrossRealmTransformWritable(port).
- To deserialize a TransformStream, we call ReadableStreamDeserialize on the first port and and WritableStreamDeserialize on the second. Then we construct a new TransformStream from the (readable, writable) pair using a new constructor for the C++ TransformStream object. No corresponding internal JavaScript stream is required.

Cross-realm identity transform

Each side of the transform needs to be created separately, in either the source realm or the target realm. The following pseudo-code describes the operations **CreateCrossRealmTransformWritable** and **CreateCrossRealmTransformReadable**.

```
function CreateCrossRealmTransformWritable(port) {
  let controller;
  let relieveBackpressure;
  let backpressurePromise = new Promise(resolve => {
    relieveBackpressure = resolve;
  });

  port.onMessage = evt => {
    const {type, value} = evt.data;
    switch (type) {
      case 'pull':
        relieveBackpressure();
        backpressurePromise = undefined;
        break;

      case 'cancel':
        controller.error(value);
        if (backpressurePromise) {
          relieveBackpressure();
          backpressurePromise = undefined;
        }
```

```
        break;
    }
  };

  return new WritableStream({
    start(c) {
      controller = c;
    },
    write(chunk) {
      await backpressurePromise;
      backpressurePromise = new Promise(resolve => {
        relieveBackpressure = resolve;
      });
      port.postMessage({type: 'chunk', value: chunk});
    },
    close() {
      port.postMessage({type: 'close'});
      port.close();
    },
    abort(reason) {
      port.postMessage({type: 'abort', value: reason});
      port.close();
    }
  }, new CountQueuingStrategy({highWaterMark: 1}));
}
```

```
function CreateCrossRealmTransformReadable(port) {
  let controller;
  let backpressurePromise = new Promise(resolve => {
    relieveBackpressure = resolve;
  });
  let finished = false;

  port.onMessage = evt => {
    const {type, value} = evt.data;
    switch (type) {
      case 'chunk':
        if (finished) return;
        controller.enqueue(value);
        relieveBackpressure();
        backpressurePromise = new Promise(resolve => {
          relieveBackpressure = resolve;
        });
        break;

      case 'close':
        if (finished) return;
        finished = true;
        controller.close();
        port.close();
        break;

      case 'abort':
        if (finished) return;
        finished = true;
        controller.error(value);
        port.close();
        break;
```

```
    }
  };

  return new ReadableStream({
    start(c) {
      controller = c;
    },
    pull() {
      port.postMessage({type: 'pull'});
      return backpressurePromise;
    },
    cancel(reason) {
      finished = true;
      port.postMessage({type: 'cancel', value: reason});
      port.close();
    }
  }, new CountQueuingStrategy({highWaterMark: 0}));
}
```

The real implementation will not be this clean because

1. It will have to protect itself against changes to the global object, and
2. It will have to clean-up after exceptions from postMessage().

Garbage Collection

Most of the responsibility for correct garbage collection is delegated to the MessagePort it uses internally. See the "Ports and garbage collection" section from the HTML standard. The explicit calls to `port.close()` when the stream is errored or closed help ensure timely garbage collection.

In the usual case, developers will drop the reference to the original stream in the source realm, and only the transferred stream will be retained. In this case the transferred stream keeps its MessagePort alive, the entangled MessagePort keeps the original stream alive, and the original stream keeps the underlying source / sink / transformer alive. On close or error, the MessagePort is closed, untangling it, and the original stream can be collected.

A common error we can expect developers to make is to keep the original stream referenced even when they don't need it any more. The stream will be collected as above once they stop referencing it.

If a developer discards the transferred stream (without closing or erroring it), but keeps a reference to the original stream, the original stream will keep the transferred stream alive. It appears that it would be safe to collect the transferred stream in this case, but it is not completely clear. In particular, the original stream needs to stay locked as if the transferred stream is still there.

If a developer discards both the original and transferred sides without closing or erroring them, the MessagePorts should be collected as unreferenced. Sadly this doesn't work: https://bugs.chromium.org/p/chromium/issues/detail?id=798855. It doesn't appear possible to fix this while still using MessagePort.

If one of the realms is destroyed (eg. a Worker is terminated), the MessagePort in the other realm will be closed, making the stream collectable. This is implemented by blink::MessagePort::ContextDestroyed().

Alternatives Considered

- See **A**, **B**, **C** and **D** under **Serialization**.
- Originally internal ports were added to the end of the existing `ports` field, and a new index field `first_internal_port` indicated the separation between ports that should be exposed to JavaScript and those that shouldn't. The semantics of `first_internal_port` were confusing and error-prone, so this approach was abandoned.
- Instead of building the transfer out of a cross-realm identity transform and pipeTo(), we could instead build it out of an identity "wrapper" around the reader or writer. This would be more complex and allows less code reuse, but may have some benefits. In particular, the cross-origin identity transform does not reflect back the results of write, close and abort calls.

Scope

Streams changes and new C++ glue in blink/renderer/core/streams.

Changes to serialisation in blink/renderer/bindings/core/v8/serialization.

Changes to messaging in third_party/blink/renderer/core/messaging.

Tests in LayoutTests/external/wpt/streams/transferable

Risk

As with anything involving threads, there is a danger of subtle bugs that are not caught during testing. This risk is mitigated in the design by the re-use of existing well-tested code paths.

The basic concept of transferring a stream is simple enough that we don't need to worry about the design changing. The main risk in this respect is that very few developers use it and we have to carry the complexity of a mostly-unused feature. We are hearing a lot of interest and enthusiasm from developers for the feature, so the risk of non-adoption appears low. We will also engage with developers from an early stage to ensure that we have feedback before too many resources are expended.

Release Plan

The feature will be developed behind a flag. In practice this means that transfer of Streams will be disallowed when the flag is not set (which then makes all the other features unreachable).

Since the feature won't do anything if it is not used, there is little benefit to a field trial. Once the feature is considered stable and approved to ship, it will be turned on by default.

Work Estimate

- Design and prototyping: 2 weeks

- Early (ReadableStream only) version landed behind a flag: 1 week
- WritableStream and TransformStream also supported: 1 week
- Making the implementation clean enough to ship: 3-4 weeks
- Creating extensive wpt tests: 1-2 months.
- Standardisation: 6 months.

Document History

Date	Who	What
27 September 2018	Adam Rice	Forked from older design doc.
11 November 2018	Adam Rice	Many updates reflecting the latest design.
19 November 2018	Adam Rice	Updated to reflect that MessageEvent and postMessage implementations are no longer modified.
29 November 2018	Adam Rice	Updated description of TransformStream transfer. Added correct backpressure handling to the CreateCrossRealmTransformReadable pseudocode.