Proposed Architecture

Overview

The Flutter DevTools Network panel currently focuses on HTTP requests, which works well for traditional client–server communication. However, many modern applications rely on long-lived connections such as WebSockets or streaming RPC systems like gRPC. These protocols behave very differently from HTTP because they maintain persistent connections and exchange messages continuously rather than following a request/response pattern.

The goal of this project is to extend the existing network profiling infrastructure so that WebSocket (and potentially gRPC) traffic can be captured and displayed in the DevTools Network panel. This will allow developers to observe message flow, inspect frame sizes, and better understand how their applications communicate with backend services in real time.

At a high level, the architecture involves three layers:

1. Capturing WebSocket activity at the runtime level
2. Exposing that activity through the VM Service protocol
3. Rendering the captured events inside the DevTools Network panel UI

The project will begin with WebSocket support and, if time allows, expand to gRPC traffic as well.

Step 1: Instrument WebSocket Traffic

The first step is capturing WebSocket traffic inside the Dart runtime.

WebSocket activity originates from the `dart:io` WebSocket implementation. The key idea is to instrument the points where messages are sent and received so that each frame can be recorded with useful metadata.

For each message, the system should capture:

- Timestamp

- Direction (sent or received)
- Frame size
- Frame type (text or binary)

Conceptually, this can be represented as a stream of frame events:

time: 12:01:21
direction: sent
size: 14 bytes
type: text

These events form the raw data that DevTools will later display.

Instrumentation can either be implemented directly in the WebSocket implementation or through a profiling wrapper that emits events when `add()` and `listen()` are called.

Step 2: Expose Events Through VM Service

Once WebSocket activity is captured, the next step is making those events visible to DevTools.

Flutter DevTools communicates with the running application through the Dart VM Service protocol. HTTP network profiling already uses this mechanism, typically by emitting timeline events or structured service events.

The WebSocket instrumentation layer can follow a similar pattern by emitting structured events through the runtime profiling system. Each frame event can be sent through the VM Service and grouped by connection.

Example event structure:

```
{
 connectionId: 12,
 timestamp: 1710000000,
 direction: "received",
 size: 32,
 type: "text"
}
```

DevTools can then subscribe to these events and reconstruct the message stream for each WebSocket connection.

Core Principles

Several principles guide this design.

First, the instrumentation layer should introduce minimal overhead. Network profiling should not noticeably impact application performance.

Second, the event format should remain consistent with existing DevTools network profiling structures where possible. This will make integration with the existing UI easier.

Third, the implementation should be incremental. WebSocket support will be implemented first, and the design should make it straightforward to add support for other protocols such as gRPC later.

Example Transformation

Input (Application WebSocket usage)

A Flutter application may interact with a WebSocket like this:

```
socket.add("hello");
socket.listen((message) {
 print(message);
});
```

Generated Network Events

Internally, these operations would produce events similar to:

```
[12:00:01] sent       5 bytes
[12:00:01] received   32 bytes
[12:00:02] received   5 bytes
```

These events can then be transmitted through the VM Service and displayed in the DevTools Network panel.

Design Decisions

One important decision is how WebSocket traffic should be surfaced through the runtime.

There are two main possibilities:

1. Emitting timeline events through `dart:developer`

2. Exposing a dedicated VM Service event stream for WebSocket traffic

Timeline events have the advantage of reusing existing profiling infrastructure, while a dedicated stream could allow more structured data and easier filtering.

Another decision involves how to represent long-lived connections in the UI. Unlike HTTP requests, WebSocket sessions may remain active for extended periods. The Network panel may therefore represent each connection as a parent entry with individual frame events nested underneath.

Benefits

Adding WebSocket support to the DevTools Network panel will significantly improve debugging capabilities for Flutter developers.

Developers will be able to:

- observe real-time message flows

- debug unexpected server responses

- analyze payload sizes and timing

- understand how persistent connections behave during application runtime

This is especially valuable for applications that rely heavily on real-time communication, such as chat systems, multiplayer games, and live dashboards.

Why This Architecture

This architecture builds directly on the existing DevTools networking infrastructure rather than introducing a completely separate system.

By capturing WebSocket events at the runtime level and exposing them through the VM Service, DevTools can consume the data in a way that is consistent with how HTTP traffic is already handled.

This approach keeps the implementation modular and minimizes disruption to the current codebase while enabling future extensions, including gRPC support and additional streaming protocols.