HW2

Link to the Code

Our codebase can be found in the following public GitHub repository, on the *main* branch:

`https://github.com/MKJM2/cs2620-wire-protocols`

Does the Use of This Tool (gRPC) Make the Application Easier or More Difficult?

Definitely easier. Serialization/deserialization seemed to us like a very "automatable" task, so having something do most of the work for us, allowing us to focus on the actual interface, is great!

Ease of Implementation & Extensibility

- **No need to write a custom parser**: Unlike our binary protocol, gRPC eliminates the need to write custom encoding/decoding logic.
- **JSON**: JSON is easy to extend—just add new fields. However, it lacks the efficiency of binary serialization.
- **Custom Wire Protocol**: Our custom wire protocol isn't necessarily hard to extend, but it's definitely annoying. For every new `op_code` we want to support, we have to implement additional encoding/decoding logic.
- **Protobuf/gRPC**: Protobuf and gRPC are super easy to extend. You just modify the `.proto` definition and recompile the code! The decoding/encoding logic is automatically handled for you. It reminds me of using `serde` in Rust.

What Does It Do to the Size of the Data Passed? Memory Usage Benchmarking

Benchmarking Against JSON and Our Custom Protocols

A challenge with benchmarking is that many of the message types have variable or unbounded lengths. For example, a client sending a message can send a message of any arbitrary length.

After some deliberation, we decided on the following approach:

- For message types (`opcodes`) where the length isn't variable, we provide a detailed size comparison in the table below.
- For variable-length messages, we test across a range of realistic scenarios and edge cases to account for variability.

Assumptions and Limitations

- **Insecure gRPC Channels**: For this analysis, we use insecure gRPC channels, so we do not account for overhead from TLS encryption or HTTP/2 headers. We only measure the size of the actual serialized Protobuf payload.
- **Layer Differences**: This analysis isn't fully fair because:
  - Operating on raw sockets (our custom protocol) works directly on **Layer 4** (Transport Layer) of the OSI model (e.g., TCP or UDP).
  - gRPC is an **application-layer protocol** (Layer 7), built on top of HTTP/2.

– By design, gRPC provides additional features like multiplexing, flow control, and built-in support for streaming, which inherently add overhead.

Because the protocols operate at different levels of abstraction, we decided to focus on **raw serialization efficiency**. Specifically, we compare: - The size of the JSON dump. - The size of our custom binary payload. - The size of the Protobuf payload sent over the network.

Testing Across Realistic Scenarios and Edge Cases

Realistic Scenarios

1. **Small Messages**:
   - Example: A simple "Hello" message.
   - Purpose: Highlights the overhead of headers/metadata for each protocol, which will comprise a large proportion of the bytes sent over the network.
2. **Medium Messages**:
   - Example: A typical chat message consisting of a few words or a single page of accounts for the `LIST_ACCOUNTS` operator.
   - Purpose: Represents the average case request sent in a chat application.
3. **Large Messages**:
   - Example: A list of hundreds of accounts or a large chat message consisting of multiple English sentences. Another example is fetching unread messages where there are 100+ unread messages.
   - Purpose: Tests how well each protocol handles large payloads and whether the overhead scales with payload size.

Edge Cases

1. **Empty Payloads**:
   - Example: Protocols that allow empty payloads. Sending an empty message to a user.
   - Purpose: Provides an estimate of the specific number of overhead bytes in each protocol.
2. **Maximum-Sized Payloads**:
   - Example: Messages with the largest allowed content for each protocol.
   - Purpose: Tests whether, in the limit, the size of the overhead becomes negligible compared to the size of the actual message contents.

Size Comparison Table for serialized messages

Message Type	Case	JSON (bytes)	Custom (bytes)	Protobuf (bytes)
LOGIN	empty	116	70	66
	small	118	72	70
	medium	127	81	79
	large	166	120	118
CREATE_ACCOUNT	empty	125	70	66
	small	127	72	70
	medium	137	82	80
	large	225	170	168
SEND_MESSAGE	empty	47	6	8
	small	50	9	15
	medium	71	30	36
	large	1267	1226	1233
DELETE_ACCOUNT	empty	44	2	6
	typical	50	2	6

Message Type	Case	JSON (bytes)	Custom (bytes)	Protobuf (bytes)
LIST_ACCOUNTS	empty	69	8	6
	small	72	10	14
	large	77	14	18
READ_MESSAGES	empty	74	7	6
	small	80	14	17
	large	116	49	52
DELETE_MESSAGE	empty	43	3	6
	small	44	7	9
	large	433	403	108
CHECK_USERNAME	empty	40	4	0
	small	42	6	4
	large	220	184	183
QUIT	empty	14	2	6

For discussion of this table consider the following: 1) JSONs overhead (in bytes) is high, due to being text-based and human readable (need to store braces, colons, field names in plaintext), 2) Our custom protocol, while performing efficient serialization of data (same as protobuf), also includes an opcode and version code. 3) Protobuf by itself (the serialization part only) is *almost* as efficient as our custom protocol, but not quite there in some scenarios (as outlined in above table). It does come with the massive advantage from implementation standpoint: considering the engineering time required to code up the application with gRPC versus writing custom binary encoding/decoding logic, we'd much rather deal with Protobuf. Lastly, we acknowledge that gRPC has a ton of overhead over Protobuf itself, by virtue of being built on top of HTTP2. This is a tradeoff for gRPC's features, and frankly, we're willing to accept it given how easy it was to integrate (A more comprehensive analysis could consider using `wireshark` or `tcpdump` to examine the actual HTTP/2 payloads sent over the network)

How Does It Change the Structure of the Client? The Server?

Switching to gRPC fundamentally alters the client by abstracting away the complexities of raw socket management and custom JSON handling. Instead of dealing with manual connection setups, message buffering, and decoding, the client now interacts with a generated stub that provides a set of clearly defined RPC methods. This means that the client can simply call functions like Login(), SendMessage(), or SubscribeEvents() without having to worry about the underlying network protocols. Also, push notifications or asynchronous events are managed through gRPC's built-in streaming mechanism. This allows the client to receive updates in a dedicated background thread without manually parsing an incoming data stream.

On the server side, gRPC transforms the architecture from a low-level, event-driven model into a structured, service-oriented framework. The server no longer has to manually read from sockets, split messages by newlines, or dispatch actions based on a JSON "ACTION" field. We instead implement each functionality as an individual RPC method defined in a Protocol Buffers file. This results in a cleaner separation of concerns where the business logic is decoupled from the network handling. The server also benefits from gRPC's concurrency model, which manages multiple client requests efficiently using a thread pool, and from the streaming API that simplifies the process of sending asynchronous notifications to connected clients.

How Does This Change the Testing of the Application?

The JSON wire protocol tests and the gRPC tests we implemented differ significantly in both approach and structure. The JSON tests focus on raw transport layer simulation, using FakeSocket classes to intercept network communication and verify serialized JSON payloads. They rely on pytest fixtures for dependency

injection and global state management, with explicit reset mechanisms between tests. But, our gRPC tests focus on the service interface rather than the transport details. Instead of mocking sockets, they mock the gRPC service stubs and interceptors, working with strongly-typed protocol buffer messages instead of raw JSON strings. The tests use unittest's setUp/tearDown pattern with patchers to isolate components. The JSON tests handle serialization/deserialization explicitly, asserting against parsed JSON content. Our gRPC tests work with structured message objects. It on the values and fields rather than the serialization format. For asynchronous operations, the JSON tests verify callback execution for events, while our gRPC tests directly examine the streaming RPC generators and subscription mechanisms. We can say that the JSON tests operate closer to the wire protocol level, while our gRPC tests operate at a higher abstraction level.