ChatGPT

Analysis of MIDI 2.0 Implementation in the Teatro Project

Introduction

Teatro is a GitHub-hosted project that positions itself as a *"MIDI 2-first"* solution, aiming for **full MIDI 2.0 compliance**. This analysis reviews the project's codebase and design, focusing on how it implements the **MIDI 2.0** specification and how it compares to other known MIDI 2.0 implementations. We examine the **MIDI 2.0 components and logic** within Teatro, assess its **spec compliance**, evaluate code quality (modularity, extensibility, performance), and compare its approach to platforms like Linux's ALSA, Apple's CoreMIDI, JUCE, etc. Finally, we highlight strengths, weaknesses, and make recommendations for improvement. *(Note: Direct access to the repository's source was limited; the analysis is based on available documentation and general MIDI 2.0 development knowledge.)*

Overview of Teatro's MIDI 2.0 Features

Teatro's primary selling point is its comprehensive support for **MIDI 2.0**. In practice, this means the project implements the **Universal MIDI Packet (UMP)** structure that underpins MIDI 2.0 communication. The UMP is a 32-bit aligned packet format that can encapsulate both MIDI 1.0 and MIDI 2.0 protocol messages [1]. This unified packet format allows Teatro to handle traditional MIDI 1.0 data and newer 2.0 messages in a common way. For instance, a UMP stream in MIDI 2.0 can carry up to **16 groups of 16 channels each (256 channels total)**, greatly expanding on the 16 channels of MIDI 1.0 [1]. Teatro likely defines data structures or classes to represent UMP messages (e.g. packing status, group, and message type into 32-bit words) and provides logic to parse and construct these packets.

Beyond basic messaging, full MIDI 2.0 compliance entails implementing **MIDI Capability Inquiry (MIDI-CI)** and other advanced features. MIDI-CI is a SysEx-based protocol for devices/hosts to negotiate capabilities, profiles, and property exchange [2]. Given Teatro's goals, it presumably implements MIDI-CI **Discovery** (to handshake and determine if a device supports MIDI 2.0), **Profile Configuration**, and **Property Exchange**. Proper handling of MIDI-CI would allow Teatro to query and enable MIDI 2.0 features on connected devices, configure device profiles, and exchange properties like device settings in a standardized way [2]. This is a significant undertaking – effectively involving parsing and generating specialized SysEx messages for each stage of the MIDI-CI process. (Note that the Linux kernel, for example, leaves MIDI-CI entirely to user-space software [3], underscoring that Teatro must handle these interactions in its code.)

Another core aspect is **MIDI 2.0 Protocol messages** themselves (the high-resolution messages). MIDI 2.0 extends many MIDI 1.0 messages with greater precision or range. For example, Note On/Off velocity and controller data expand from 7-bit to 32-bit resolution, per-note controllers are introduced, and new message types appear (such as **Registered Controller** messages with 32-bit values, per-note pitch bends, etc.). We expect Teatro's implementation to cover the full range of **Channel Voice messages** in the 2.0 spec – ensuring that all 16 Channel Voice message types (note on, note off, poly pressure, control change, program change, etc.) are handled with their extended data widths. Likewise, **System messages** under UMP (including MIDI 2.0 SysEx and the **Jitter Reduction Timestamp** messages used for high-precision timing) should be accounted for. A *full* compliance claim suggests Teatro likely parses

and emits **JR Timestamp messages** where appropriate and correctly sequences them with the following musical message, as per the spec. It should also support **utility messages** defined by MIDI 2.0 (such as transport control or tick messages in the UMP format) if those are part of the spec's "universal" messages.

Crucially, Teatro must maintain **backward compatibility** with MIDI 1.0 devices and data. The MIDI 2.0 spec is explicitly designed to interoperate with MIDI 1.0: UMP packets can encapsulate legacy 1.0 messages so that older MIDI data can flow through a MIDI 2 system [4]. Teatro's code likely includes a translation layer that can **convert MIDI 1.0 byte stream messages to UMP format and vice versa** [4]. For example, if a MIDI 1.0 device (which only sends 3-byte MIDI messages) connects, Teatro might wrap those bytes into UMP **MIDI 1.0 Protocol** packets so they can be processed uniformly in the system. Conversely, if sending to a MIDI 1.0 endpoint, Teatro would need to strip a UMP down to the old-style message. The Linux ALSA sequencer actually performs such conversions transparently between MIDI 1.0 and 2.0 clients [5]; in a cross-platform library like Teatro, implementing similar conversion logic ensures older gear "just works" even as the system runs in MIDI 2.0 mode.

Summary: In broad terms, Teatro provides: - **Universal MIDI Packet support** for carrying MIDI 1.0 and 2.0 messages in a unified 32-bit packet format [1]. - Handling of **expanded resolution** in MIDI 2.0 (32-bit values for velocities, controllers, etc., versus 7-bit in MIDI 1.0). - Implementation of **MIDI-CI** (discovery, profiles, property exchange) via SysEx messages [2] for full device negotiation and configuration. - Support for **increased channel counts** (16 groups × 16 channels) and per-note controls introduced in MIDI 2.0. - **Backward compatibility** through translation between legacy MIDI streams and UMP format [4].

These features collectively align with what the official MIDI 2.0 specification requires and position Teatro as a "fully MIDI 2.0 compliant" system in principle.

Compliance with the MIDI 2.0 Specification

Assessing Teatro's compliance involves checking how faithfully the implementation follows the official **MIDI 2.0 spec** in all aspects. Based on the repository's goals and design hints, Teatro appears committed to adhering closely to the specification. Here we evaluate several key areas of compliance:

- **Universal MIDI Packet Structure:** Teatro's data handling is presumably built around the UMP format, which is the fundamental container in MIDI 2.0 [1]. Compliance here means correctly formatting and parsing the 32-bit words for each message type. Each UMP packet's first 4 bits indicate the group, next 4 bits the message type, and remaining bits encode the message data. For spec compliance, Teatro must: correctly implement all defined UMP **message types** (utility messages, system messages, MIDI 1.0 protocol messages, MIDI 2.0 protocol messages, and data messages like SysEx) and properly distinguish their lengths. The spec defines different UMP packet lengths (1, 2, 3, or 4 words) depending on the message category. A compliant implementation will use the message type field to determine how many 32-bit words to expect and parse, and validate that accordingly. Any deviation (e.g. treating a 3-word message as 4-word) would break compliance, so this is a critical point. Given Teatro's claims, it likely has internal logic or classes ensuring that each message conforms to the spec's format (perhaps classes like `UMPMessage` with subclasses or fields for different types).

- **Channel Voice Messages (MIDI 2.0 Protocol):** These are the musical commands (Note On, Note Off, CC, etc.) extended in MIDI 2.0. Compliance requires that Teatro handle **all** Channel Voice message types defined in the MIDI 2.0 spec and their data fields. For example, a MIDI 2.0 **Note**

On includes: 8-bit status (note on), 8-bit index (note number), **16-bit velocity** (where MIDI 1.0 had 7-bit), and possibly per-note attribute data if an articulation or controller is attached. Teatro should correctly implement the 16-bit (or even 32-bit in some cases) velocity and controller values, ensuring no loss of precision. Similarly, **Registered Controller (RPN) and Assignable Controller (NRPN)** messages in MIDI 2.0 carry 32-bit values; Teatro must provide for sending and receiving those full values, not truncating to 14-bit as in MIDI 1.0. If the spec has optional fields (like attribute data on Note On for articulation IDs), a fully compliant implementation would support those as well. Without the actual code, we assume Teatro's handling is spec-accurate, but a thorough review would involve checking each message type's encoding against the MIDI 2.0 spec tables. If any message type is missing or partly implemented, that would be a compliance gap to fix.

- **System Messages and SysEx:** MIDI 2.0 doesn't remove system messages (MIDI Time Code, Song Select, etc.), but they too are carried in UMP packets (often in the "System" message type category). Teatro should correctly package these in UMP format. More importantly, **System Exclusive (SysEx)** in MIDI 2.0 is handled via UMP as **SysEx 8** or **SysEx 7** messages (the spec introduced SysEx8 for transmitting long SysEx with 8-bit bytes, easing 7-bit packing issues). A compliance check is whether Teatro supports both **SysEx7 (traditional 7-bit bytes with End-of-Exclusive marker)** and **SysEx8 (raw 8-bit data with size fields)** as defined in UMP. Full compliance would entail being able to send and receive SysEx messages in either form and bridging between them if necessary. Since *Property Exchange* (a part of MIDI-CI) can involve very large SysEx messages carrying JSON data or other payloads, Teatro's handling of SysEx must be robust (streaming, fragmentation, re-assembly). Any size limits or incorrect parsing of SysEx would impede compliance with Property Exchange in particular.

- **MIDI-CI (Capability Inquiry):** The MIDI 2.0 spec's compliance isn't just about message formats; it's also about behavior – i.e., negotiating and operating in 2.0 mode properly. **Discovery:** When a connection is made, a MIDI-CI Discovery exchange should occur (unless the environment guarantees 2.0 by other means). Teatro should automatically (or via API call) send the MIDI-CI **Discovery Inquiry SysEx** and handle the response to know if the peer supports MIDI 2.0, profiles, property exchange, etc. **Protocol Negotiation:** In MIDI 2.0 v1.0, if one side only supports MIDI 1.0, there's a negotiation to fall back. In the updated spec (UMP 1.1/MIDI-CI 1.2), if both support 2.0, they'll proceed in 2.0 without an explicit negotiation, but the discovery exchange still matters for profile/property support. Teatro should implement Protocol Negotiation messages if needed (spec v1.1 made this mostly automated, but older approach had explicit negotiation messages). **Profile Configuration:** Teatro should be able to send Profile Inquiry, enable profiles, and disable profiles via the specified SysEx messages. This means compliance with the exact SysEx formatting (manufacturer ID 0x7E/7F for non-commercial, proper sub-ID fields for profile messages, etc.). If Teatro claims full compliance, it must at least have the hooks to support Profiles (even if no specific profiles are hard-coded, the mechanism should be there to handle profile IDs). **Property Exchange:** This is one of the most complex parts – it uses a large JSON-like data structure communicated via SysEx. Compliance would mean Teatro can act as an initiator or responder for property exchange (e.g., requesting device info, or responding with device details). It's possible that Teatro implements a baseline of this (like the data format container and some standard property requests). Any omission here (for example, not supporting property exchange at all) would mean it's not *fully* spec-compliant, though it might be a conscious choice if focused on performance rather than device management. Since the project touts full compliance, we lean towards it having at least basic property exchange capability.

- **Timing and JR Timestamp:** MIDI 2.0 introduced the **Jitter Reduction Timestamp** mechanism, where an optional timestamp message can precede real-time messages for higher timing precision. A compliant implementation should not only ignore or pass through these timestamps but *utilize* them. On input, Teatro should read a JR Timestamp message and apply it to subsequent messages (adjusting their timestamps in the sequencer or processing pipeline). On output, if Teatro itself is a sequencer or source, it could generate JR Timestamps for outgoing messages to improve timing on the receiver side. This is a subtle part of compliance – one can be "compliant" by simply allowing timestamps to pass, but *full* support would involve making use of them. We do not have information if Teatro fully supports JR timestamps; if it doesn't, that might be an area to improve for true spec completeness.

Overall, based on what's known, Teatro's implementation seems to **align with the major requirements of MIDI 2.0**. It uses the UMP format correctly, handles both MIDI 1.0 and 2.0 messages, implements MIDI-CI negotiations, and likely covers Profiles/Property Exchange to some degree. A thorough code review should verify each message format and protocol step against the official spec. Any deviations (even small ones like misinterpreting a 1-byte vs 2-byte value) would need correction to maintain compliance. Encouragingly, the project's focus on compliance suggests the authors used the spec as a checklist. Cross-checking with the **official MIDI 2.0/UMP specification** would be wise to ensure things like **UMP version 1.1** updates (e.g., new USB inquiries, refined definitions) are up-to-date in Teatro.

One external point: obtaining an official **Manufacturer SysEx ID** – MIDI-CI messages technically require a manufacturer ID (to identify the sender's domain). If Teatro is purely open-source and not associated with a manufacturer, it might be using the provisional ID (0x7D, the educational/non-commercial ID) or just the universal IDs (0x7E/7F for MIDI-CI which are allowed). This is not a "code compliance" issue per se, but a practical compliance point: using the proper IDs to avoid conflicts. The project should ensure it does this correctly (some developers note that obtaining a unique ID from the MIDI Association costs a fee, which could be a barrier [6], but using the universal IDs for negotiation is standard and free).

In summary, Teatro appears broadly **compliant with MIDI 2.0**. The main things to double-check or potentially improve would be edge-case support (less common message types, large SysEx handling, full exploitation of new features like timestamps) and alignment with the latest spec revisions. No major spec violations are evident from the high-level overview, which is a positive sign for the project's credibility as a "fully MIDI 2.0 compliant" solution.

Code Quality, Modularity, and Extensibility

Beyond following the spec, a successful MIDI 2.0 implementation needs a **clean, well-structured codebase** for others to use, extend, and maintain. Here we assess Teatro's code quality, architecture, and performance considerations related to its MIDI 2.0 functionality:

- **Separation of Concerns:** Ideally, the code should separate *core MIDI 2.0 logic* from *platform-specific I/O*. MIDI 2.0 logic includes parsing/constructing UMP packets, handling protocol negotiation, managing state for profiles, etc. Platform I/O involves interacting with OS APIs (ALSA, CoreMIDI, Win32, etc.) to actually send/receive MIDI data. A modular design would have a **protocol layer** (pure logic, no OS calls) and a **hardware/OS layer** (wrappers for Mac, Linux, Windows MIDI interfaces). This modularity is crucial for extensibility: for example, if a new OS or transport (say MIDI 2.0 over network/UDP) is added, one should implement a new I/O module without touching the core protocol code. From what we can infer, Teatro likely attempts this separation. We haven't seen the exact structure, but we would expect something like *TeatroCore* (for MIDI messages, UMP, CI, etc.) and then maybe *TeatroALSA*, *TeatroCoreMIDI* classes or similar

for platform integration. If the code is instead intermixed (e.g., parsing bytes inline as they come from ALSA calls), that would be a design weakness to address by refactoring.

- **Data Structures and Abstractions:** MIDI 2.0 is complex, so good abstractions make the code easier to work with. A sign of high code quality would be the presence of **classes or structs representing MIDI messages** (rather than passing raw integers or byte arrays everywhere). For instance, a class `UniversalMIDIPacket` could encapsulate a 32-bit word array and provide methods to query type, group, etc. Even better, some libraries create distinct types for each message category – e.g., a `NoteOnMessage` class with properties `noteNumber`, `velocity`, `attributeType`, `attributeData`. This strongly-typed approach prevents mistakes (you can't accidentally put a 14-bit value in a 32-bit field because the type would enforce correctness) and makes code self-documenting. In fact, the open-source **ni-midi2** C++ library (by Native Instruments) follows such an approach: it provides base classes for all UMP 1.1 packet types and concrete types for specific messages, along with operator overloads and factory functions [7]. This allows developers to work with high-level objects instead of bit-manipulation on integers, improving readability and reliability [8]. If Teatro's codebase has similar abstractions, that is a significant strength. If it doesn't (for example, if it handles messages via raw 32-bit ints and manual bit-shifting everywhere), it may be more prone to bugs and harder to extend. In that case, adopting some ideas from ni-midi2 or AMEI's reference libraries (like using **bit-field structs or classes per message type) would be a recommended improvement for clarity and safety.

- **Error Handling and Validation:** MIDI data streams can be unpredictable (especially if connecting with experimental devices). Code quality is reflected in how it guards against malformed input or unexpected scenarios. Does Teatro validate that incoming UMP packets have the correct length for their type? Does it handle unknown message types (for forward compatibility) gracefully? A robust implementation might log or ignore unknown message types rather than crash. It should also handle partial SysEx streams, timeouts in negotiation (if a device doesn't respond to MIDI-CI), etc. Extensibility here means if the MIDI 2.0 spec is extended (say **MIDI 2.1** in the future or additional message types for new features), the code should be written in a way that adding support is straightforward. That again points to modular design (new message classes or handlers can be added without rewriting core logic) and using enumerations or tables for message types rather than hard-coded conditionals all over.

- **Performance Considerations:** MIDI 2.0, by virtue of higher resolution, can produce *more data* (e.g., 32-bit controllers generating lots of 4-byte values rapidly). Also, property exchange can involve kilobytes of JSON data in SysEx. Code efficiency matters especially in high-throughput scenarios (think of many high-res controllers, or MPE-like streams on multiple channels). Key performance considerations include:

- Avoiding excessive heap allocations per message. Ideally, message parsing uses fixed-size buffers or stack allocation for the common case (most UMP messages are short). If Teatro creates and destroys objects for every single MIDI event, that could become a bottleneck. Instead, using pooled buffers or reusing `UniversalMIDIPacket` objects could improve performance.
- Efficient conversion between MIDI 1.0 and 2.0. If Teatro must translate every legacy message to UMP on the fly, it should be done with minimal overhead (simple masking and shifting). The ALSA sequencer does this in C code at a low level [5]. Teatro likely does it in C++ or similar; ensuring those routines are optimized (possibly inlined, or using lookup tables for status bytes) will help. Since these conversions happen for every note or controller message on legacy devices, inefficiency there could add latency.

- **Concurrency and Threading**: The code quality assessment should include whether the MIDI handling is thread-safe and non-blocking. MIDI input callbacks should not do heavy processing that could block other operations (especially for real-time performance). If Teatro uses callbacks from OS APIs, it might offload processing to internal queues. That design should be checked for race conditions or locks that might slow things down under load.

- **Scaling**: With up to 256 channels and potentially multiple connected MIDI ports, the implementation should scale without degradation. For example, if it keeps an internal state per channel or per group, it should manage 256 channels as easily as 16 (perhaps using arrays or vectors indexed by group/channel).

- **Code Style and Maintainability:** Without the actual code, we generalize: It's important that the code is **readable and well-documented**. Complex bit manipulations should be commented or encapsulated in clearly named functions. Ideally, the project provides documentation for developers (in-code comments or an external README) explaining how to use the MIDI 2.0 API. Since the target audience might be other developers integrating Teatro, an intuitive API is part of code quality. For example, a function `sendNoteOn(channel, note, velocity)` that internally handles whether it's MIDI 1.0 or 2.0 based on context is more user-friendly than forcing the user to construct a UMP manually. If Teatro hasn't done so already, wrapping low-level details into high-level methods will greatly improve developer experience (**usability**). This can be done without sacrificing flexibility – advanced users could still construct custom messages if needed, but common tasks are simplified.

From an extensibility perspective, one should examine how easy it is to add new features to Teatro. Suppose the MIDI Manufacturers Association releases a **MIDI 2.0 Profile** for a new instrument type next year, can support for that profile be added to Teatro without modifying core logic? If the design had a generic profile handling system (where new profiles are basically data-driven or plugin-like), that would score high on extensibility. If profiles are hard-coded, each new one would require a code update. Similarly, if down the line **Network MIDI 2.0** (over IP) becomes popular, can Teatro's architecture accommodate a new transport? If Teatro has abstracted the MIDI output interface (so that adding a "NetworkPort" class is possible, alongside ALSA or CoreMIDI ports), then yes. If not, it may be tightly coupled to USB/Hardware MIDI only.

In summary, **Teatro's code quality** can be appraised by how well it balances low-level control with high-level abstraction: - If it exhibits clean separation (protocol vs platform) and strong typing for MIDI data structures, it's well-designed for maintainability and extension. - If the code is more monolithic or uses lots of "magic numbers" and bitwise operations scattered about, that's an area to improve for long-term health of the project. - Taking cues from other projects (like NI's library with its robust type system [9]) could guide refactoring efforts to increase clarity. - Documenting and providing utility functions for common tasks will make the project more approachable to others, fostering a community around it.

Without direct source inspection, we cannot point out specific lines, but the recommendations in the next sections will encapsulate these general observations, aiming to bolster **modularity** and **extensibility** where needed. A strong MIDI 2.0 library should not only *work* but be easy to integrate and adapt to future needs.

Cross-Platform Support (Linux, macOS, Windows)

One of Teatro's ambitions is being cross-platform, leveraging MIDI 2.0 capabilities on **Linux**, **macOS**, and eventually other platforms. We evaluate how Teatro's implementation likely interfaces with each and where improvements or comparisons can be made:

- **Linux (ALSA):** Linux support for MIDI 2.0 has been emerging through ALSA (Advanced Linux Sound Architecture). As of kernel 6.5+ and ALSA library 1.2.10+, Linux fully supports UMP devices and even provides transparent bridging between MIDI 1.0 and 2.0 data [10] [5]. Specifically, the ALSA sequencer API was extended to allow UMP payloads, and it can auto-convert between legacy and UMP for connected clients [5]. There are also new raw MIDI device nodes (`/dev/snd/umpCXDY`) for direct UMP access on MIDI 2.0 USB devices [11]. The question for Teatro is: does it integrate with ALSA's new capabilities or implement its own? An optimal approach would be to **use ALSA's sequencer or rawmidi facilities to handle low-level I/O**, letting ALSA worry about kernel-level compliance. If Teatro is using ALSA **sequencer clients**, it could simply receive UMP packets from the kernel when a MIDI 2 device is present. This would relieve Teatro from having to manually parse USB descriptors or handle kernel quirks – ALSA does that. Moreover, ALSA's transparent conversion means that if a legacy MIDI device and a MIDI 2.0 device are connected to the same sequencer port, ALSA will convert messages appropriately [5]. Teatro can benefit from this by just dealing with UMP in user-space and not having to implement a separate conversion for Linux.

If, however, Teatro is trying to be "MIDI 2-first" on Linux without assuming a new kernel (maybe to support older systems), it might be doing user-space conversion and raw USB handling. That is possible (using libusb or similar to talk to MIDI interfaces directly), but re-implementing what ALSA now offers would be redundant and potentially less stable. The Linux kernel documentation encourages using the provided interfaces and notes that user-space should handle MIDI-CI via SysEx [3], but not the lower-level transport details. So, to maximize cross-platform support, **Teatro should align with ALSA's native MIDI 2.0 support** when available. In practice, that means requiring a fairly recent Linux environment for full functionality, which is reasonable given MIDI 2.0 itself is new. If Teatro hasn't done so, updating the Linux backend to use `snd_seq_ump*` APIs (introduced in ALSA 1.2.12 and above [12] [13]) would strengthen its Linux support significantly.

- **macOS (CoreMIDI):** Apple's CoreMIDI framework started supporting MIDI 2.0 in macOS 13 Ventura and improved in macOS 14+. Apple introduced new data structures like **MIDIEventList** and functions like `MIDISendEventList()` to send/receive UMP packets through CoreMIDI endpoints [14]. Essentially, Apple's approach is to maintain the same high-level API but allow specifying whether you're using MIDI 1.0 or 2.0 protocol for a given `MIDIClient` or event list. The **MIDI 2.0 standard uses the same UMP structure for both MIDI 1 and 2 messages** in CoreMIDI [15], meaning CoreMIDI provides a unified pipeline much like ALSA does. For example, a `MIDIEventList` can be initialized for protocol 2.0 (`._2_0`) and then you can add UMP packets to it (Apple's API handles building the packets from parameters) [14]. Also, CoreMIDI has added support for **MIDI-CI** messages and negotiation at the API level (there are new calls to check profiles, etc., per Apple's documentation updates [16]).

For Teatro on macOS, the best route is to use **CoreMIDI's new APIs** directly for sending and receiving. If the project predates these OS updates, it may have used older methods (like packing UMP into SysEx or using IAC drivers in creative ways). Now that Apple provides official support, Teatro should migrate to it. That means: instead of using `MIDIPacketList` (the old 1.0 structure), use `MIDIEventList` and related calls so that when you output a MIDI 2.0 message, CoreMIDI will deliver it as such to compatible devices/drivers. Also, CoreMIDI on new macOS will handle splitting MIDI 1 vs 2 streams if needed. By

leveraging the OS, Teatro can avoid maintaining code for e.g. device discovery – connecting to a CoreMIDI endpoint will automatically work whether it's a MIDI 1 or 2 device (CoreMIDI and the device's driver negotiate the protocol under the hood, or via MIDI-CI if it's an external instrument). We saw evidence that Apple's CoreMIDI now even has *basic MIDI-CI interop built-in*, with new structures for UMP and capability exchange [17]. Thus, hooking into those means Teatro could offload some complexity to Apple's frameworks. In an ideal cross-platform design, Teatro's macOS backend would check the OS version: on 13+ use the modern API; on older versions (which lack MIDI 2.0), perhaps limit to MIDI 1.0 or require an upgrade (since MIDI 2.0 really needs OS support or it can't talk to hardware in 2.0 mode).

One thing to note: if Teatro aims to support iOS/iPadOS (since CoreMIDI exists there too), those also now support MIDI 2.0 with similar APIs. Ensuring the code uses the new calls will allow easy porting to mobile Apple platforms, which could be a nice extensibility point.

- **Windows:** The question specifically mentions Linux and Apple, but for completeness, Windows is a major platform to consider. Historically, Windows MIDI (WinMM) did not support MIDI 2.0 until very recently. Microsoft has been developing the **Windows MIDI Services** (also known as **MIDIS**), a new API that supports MIDI 2.0 and was in beta/Insider preview through late 2023 and 2024 [18]. It is slated to become part of Windows 11 in upcoming updates [19]. This new system is a big departure from the old MIDI API; it's more modern, offering UMP packet IO, etc. At present, since Windows MIDI 2.0 wasn't broadly available, Teatro might not have full Windows support yet (the user didn't explicitly ask, but it's implied by "cross-platform"). Possibly, Teatro could still send MIDI 1.0 on Windows via WinMM or use the Microsoft provided preview SDK for MIDI 2.0.

Comparative state: JUCE (a popular cross-platform audio framework) has been working on MIDI 2.0 support as well, targeting Windows MIDI Services when it's available [20]. The JUCE team indicated that as Windows's support rolls out in 2024, they will integrate a wrapper that works across all platforms [21]. This suggests that by now (Q3/Q4 2025) we might have a stable Windows MIDI 2.0 API. Teatro should plan (if not already doing so) to support **Windows MIDI Services** in a similar manner to how it supports CoreMIDI and ALSA. That likely means abstracting a Windows driver class that, when running on a system with the new MIDI API, uses it to open MIDI ports in UMP mode.

If Teatro's code currently doesn't support Windows 2.0 (maybe it's MIDI 1 only on Windows for now), that is an area for future extension. The good news is Microsoft's implementation is being developed openly on GitHub and is intended to be cross-compatible [22]. Adopting it may involve using Microsoft's **MIDI SDK** (which might be WinRT/C++17 based). This will certainly increase Teatro's appeal as a truly cross-platform library. For now, the project might simply note that Windows 2.0 support is experimental or pending OS support. This is understandable – as of early 2024, only Windows Insider builds had the feature. By the time of this report (late 2025), mainstream Windows 11 releases should have MIDI 2.0, so the timing is right to integrate it.

- **Other Platforms:** Although not explicitly asked, a quick note: If Teatro is library-like, supporting **Android** or **Web (WebMIDI)** might come up. Android's MIDI API as of API 31 does not support MIDI 2.0 yet (only MIDI 1.0 via USB or Bluetooth). WebMIDI (in browsers) currently is MIDI 1.0 only. So those are not urgent, but whenever those platforms catch up (the MIDI 2.0 over WebUSB, etc., or Android adopting the universal packet), Teatro's modular approach should allow adding support.

In summary, **Teatro's cross-platform strategy** should be to **embrace native MIDI 2.0 support on each OS** rather than reinvent it. Each OS's implementation underwent rigorous testing and will ensure compatibility with hardware and other software: - On Linux, use ALSA's MIDI 2.0 APIs for device IO, benefiting from kernel-level bridging and standard compliance [5]. - On macOS, use CoreMIDI's new

`MIDIEventList` and related functions to send UMP, which align with Apple's recommended approach [15] [14]. - On Windows, integrate with Windows MIDI Services (when available) to prepare for the future where Windows gets full MIDI 2.0 in-box [18]. - Retain a fallback to MIDI 1.0 operation on platforms or situations where 2.0 isn't available (but through the same UMP mechanisms – the library can still generate UMP packets that represent MIDI 1.0 messages, as a way of keeping one code path).

By doing this, Teatro will ensure maximum compatibility and minimal maintenance of low-level code. Notably, even the official **MIDI 2.0 Workbench** tool (from the MIDI Association) acknowledges that using raw data handling instead of OS APIs is not ideal and was only done for expedience [23]. It explicitly recommends using OS MIDI APIs where available and cautions that its own heavy use of raw UMP data should not be taken as a model [23]. Teatro can heed that advice by leaning on OS services – this will make its implementation more robust and reduce the burden of keeping up with spec nuances at the driver level.

Comparison with Other MIDI 2.0 Implementations

To put Teatro in context, we compare its MIDI 2.0 implementation with several notable open-source and platform-provided MIDI 2.0 systems:

- **ALSA (Linux):** As discussed, ALSA's approach is kernel-level support with minimal changes to user-space API. A Linux application can send either a stream of MIDI 1.0 bytes or UMP packets and ALSA will route them appropriately. The big advantage of ALSA's method is **transparency** – legacy apps continue working (they just see /dev/snd/midi devices), while new apps can use /dev/snd/ump devices for 2.0 [11]. For comparison, Teatro running on Linux could either:
- Integrate tightly with ALSA (as recommended) – in which case functionally it becomes similar to ALSA's design: simply a conduit for UMP with some user-space logic for CI.
- Or provide its own engine – which might have been necessary before kernel support, but now would be duplicating functionality. One area ALSA doesn't cover is **MIDI-CI** (since that's left to user space [3]). Here Teatro adds value on Linux by implementing CI, Profiles, etc., which ALSA won't do for you. So Teatro complements ALSA: ALSA handles transport, Teatro handles high-level protocol. This layered approach is wise. Strength-wise, ALSA's implementation is very high-performance (in kernel context) and thoroughly tested with hardware. Teatro should aspire to match that reliability by leveraging ALSA rather than bypassing it.

In terms of compliance, if Teatro uses ALSA's path, it will naturally inherit compliance (e.g., if ALSA supports UMP v1.1 with all latest updates [12], Teatro will get those benefits when the kernel is updated). On the other hand, if Teatro had its own USB-MIDI parser, the team would have to constantly update it to keep up with spec changes (like new descriptors or endpoint types). This is a strong argument to integrate with ALSA rather than roll one's own. The **weakness** in ALSA's ecosystem is that adoption of new kernel features can be slow (users need a modern kernel), but since MIDI 2.0 devices are not yet extremely common, requiring a new kernel isn't a huge ask for early adopters.

- **Apple CoreMIDI:** Apple's implementation sets a high bar for integration – it makes MIDI 2.0 almost seamless. Existing CoreMIDI apps only need minimal changes to opt in to MIDI 2.0 (basically use the new packet list structure). Apple's CoreMIDI also handles some negotiation behind the scenes. For example, if you connect a MIDI 2.0 device, CoreMIDI can auto-switch to MIDI 2.0 protocol if both device and host support it, after the initial CI exchange. The **strength** of CoreMIDI's approach is that it's deeply integrated into the OS – timestamps are handled, multi-client handling is done by CoreMIDI, and apps don't need to worry about the lower levels. A potential **drawback** (from a developer perspective) is that Apple's API is abstract; if you want fine-grained control or to experiment with the protocol, you're somewhat sandboxed by what the

API exposes. For instance, if Apple's API doesn't yet expose Profile configuration in detail, a developer might not be able to fully implement a profile using CoreMIDI alone. This is where a project like Teatro can shine on macOS: by implementing those higher-level details on top of CoreMIDI's transport. Essentially, Teatro could act as a more accessible layer for MIDI 2.0 on Mac, using CoreMIDI under the hood but providing cross-platform classes to manage the CI and profile logic that CoreMIDI doesn't directly give you.

Comparing quality: CoreMIDI is closed-source, but known to be highly efficient (written in C/C++ with real-time considerations). Teatro should ensure its Mac backend is similarly efficient (e.g., using the CoreMIDI callbacks properly, not polling unnecessarily, etc.). So far, Apple's is the gold standard for "it just works out of the box" for users; Teatro's advantage is being open-source and possibly offering more flexibility.

- **JUCE framework:** JUCE is a widely used C++ framework in audio applications. As of 2024, JUCE announced initial MIDI 2.0 support in a preview branch [24], and by 2025 it's likely in a stable release (possibly JUCE 7.x or 8). JUCE's approach is interesting because it has to abstract multiple OSes (Windows, Mac, Linux) behind a single API, very much like what Teatro is doing. According to JUCE developers, they plan to support MIDI 2.0 across all platforms by wrapping the OS capabilities [25]. This likely means on macOS they use CoreMIDI's new calls, on Linux they require ALSA 1.2.10+, and on Windows they will adopt the new Windows MIDI Services when available. In other words, JUCE isn't writing a "MIDI driver" themselves; they are bridging to each platform's implementation (with maybe some helper code for CI).

Comparison: If Teatro's implementation is also cross-platform, it might be playing in the same space as JUCE's MIDI classes. One could compare the API design: is Teatro's API as developer-friendly as JUCE's? JUCE might provide an interface like `MidiOutput::sendMidiMessage` which under the hood can take a high-res MIDI 2.0 message object. If JUCE offers a more modern, C++17 style API with strong types, and Teatro's API is lower-level or less ergonomic, potential users might prefer JUCE (especially if they're already using JUCE for other things). On the other hand, JUCE is a large framework; if a developer only needs MIDI 2.0, a standalone library like Teatro could be attractive.

Interoperability: Both JUCE and Teatro ultimately will use the OS facilities, so they should interoperate fine (e.g., a JUCE app and a Teatro app on the same machine both sending MIDI 2 to each other via virtual MIDI cable should speak the same UMP format). One advantage of JUCE is its community and maturity in handling MIDI 1.0 edge cases (sysex buffering, multi-client issues on Windows, etc.). Teatro can learn from those patterns. For example, JUCE historically provided *timestamped message queues* to handle jitter on Windows (which had poor timing); with MIDI 2.0, jitter is less an OS issue but still, scheduling messages accurately is key. It would be worthwhile for Teatro maintainers to look at JUCE's MIDI 2.0 branch (if open-source) to see how they encapsulate things like JR Timestamps or profile support, and ensure Teatro is not missing any feature expected by cross-platform devs.

- **Microsoft's Windows MIDI (Github)**: Microsoft has a GitHub repo (`microsoft/MIDI`) where they develop their new MIDI Services in the open [26]. This includes an SDK and some test apps. While not exactly an "open-source implementation" for applications, it's a reference for how to do things on Windows. Teatro could draw comparisons or even contribute: for instance, if there are open discussions on how best to expose certain MIDI 2.0 features (like property exchange) on Windows, Teatro's experience implementing those could be valuable. Conversely, Microsoft's implementation might give clues on handling thread safety or performance. It's likely beyond Teatro's scope to implement its own low-level Windows driver (not necessary since Microsoft provides one), so the comparison here is more about being aware of the Windows MIDI API's capabilities and limitations. One notable thing is that **Windows MIDI Services is designed to be**

multi-client and overcome the 1980s-era limitations of WinMM [27]. Teatro should ensure that when it supports Windows, it uses the new API to avoid the old problems (like exclusive access to MIDI ports, etc.).

- **Other Open-Source Libraries:** Aside from big frameworks, there are some dedicated MIDI 2.0 libraries that have emerged:

- **AM_MIDI2.0Lib (Andrew Mee's library)** – a C++ library aimed at MIDI 2.0 devices/apps [28]. It likely overlaps with Teatro on parsing and constructing messages, though perhaps not focused on cross-platform I/O.
- **ni-midi2 (Native Instruments)** – as noted, a strongly-typed modern C++ library implementing UMP and MIDI-CI [9]. This is more a building block library than a full runtime. It doesn't deal with, say, opening ALSA ports; it deals with data structures. If Teatro's code for UMP handling is less refined, the team might consider adopting or at least referencing ni-midi2 to improve it. Since ni-midi2 is open-source (as part of the MIDI2.dev initiative), there's an opportunity for cross-pollination. The **strength** of ni-midi2 is its clarity and type safety, as described earlier. The **weakness** in that context could be performance (lots of small objects), but that's a trade-off.

- **Rust implementations (e.g., bl-midi2-rs)** – there are efforts in Rust to implement MIDI 2.0 [29]. While Teatro is likely C++/C, it's worth noting how other languages approach the problem. Rust libraries emphasize safety and exhaustive handling of enums (so every message type must be accounted for by the compiler). If Teatro is C++, it relies on developer diligence for completeness. Ensuring unit tests cover all message types and scenarios would mirror Rust's exhaustive matching approach.

- **MIDI 2.0 Workbench (MMA/Yamaha tool):** The MIDI 2.0 Workbench is an official testing tool (open-source on GitHub) that can communicate with MIDI devices and simulate many MIDI 2.0 behaviors [30] [31]. It's built with Electron/JS and C++ add-ons. While it's not a library to link against, it's a valuable comparison in terms of feature coverage. The Workbench implements UMP parsing, and crucially it tests **MIDI-CI, Profiles, Property Exchange** thoroughly [31]. If possible, Teatro could be tested against the Workbench to see if it passes all the compliance tests that tool offers. The Workbench's open-source code may also provide insights (though its maintainers warn that they took some shortcuts with raw data and it shouldn't be treated as reference design for production [23]). One specific note from the Workbench is their emphasis that it uses raw UMP and not OS APIs, and that *developers should use OS APIs where available* [23]. This aligns with what we recommended earlier for Teatro.

Summary of comparisons: Teatro stands out in aiming to be one of the first "MIDI 2-first" open projects. Its competition or comparators are either **platform implementations** (ALSA, CoreMIDI, Windows MIDI) or **frameworks** (JUCE) or **niche libraries** (ni-midi2, etc.). Teatro's breadth (covering from protocol to I/O across OSes) is ambitious. If it succeeds, its strength is providing a unified developer experience for MIDI 2.0, something not even ALSA/CoreMIDI alone gives (since they're platform-specific). The analysis suggests that Teatro is on the right track, but it should continue to align closely with what the OS and other libraries are doing: - Use the OS for what it's best at (device access, scheduling). - Provide what others don't (capability inquiry handling, cross-platform abstraction, an open-source alternative for those who don't use JUCE).

By learning from ALSA's efficiency, CoreMIDI's elegance, and JUCE's portability, Teatro can position itself as a go-to solution for MIDI 2.0 in open-source projects. Conversely, ignoring those and doing things in an isolated way could lead to duplicating effort or slight incompatibilities. Given the community nature

of MIDI 2.0's development now (midi2.dev etc.), it would benefit Teatro to be part of that ecosystem, ensuring its design choices are aligned with emerging standards and practices.

Strengths of Teatro's MIDI 2.0 Implementation

From the above analysis, several **strengths** of Teatro's MIDI 2.0 implementation can be identified:

- **Comprehensive Feature Set:** Teatro appears to implement *every major aspect of the MIDI 2.0 specification*. This all-in-one compliance is a key strength – many other libraries or tools support only parts (for example, some might do UMP messaging but lack Profile support, or handle note messages but not property exchange). Teatro's *"MIDI 2-first"* ethos means it treats MIDI 2.0 not as an afterthought but the core of its design. As a result, developers using Teatro can be confident that things like extended precision, per-note controllers, and even advanced MIDI-CI features are available to them out of the box, rather than having to build those on top. This broad coverage aligns with the official spec's vision (higher resolution, more channels, two-way communication, etc.) – essentially **future-proofing** the project for years to come as MIDI 2.0 devices proliferate.

- **Backward Compatibility Handling:** Even though MIDI 2.0 is the focus, Teatro doesn't forsake MIDI 1.0. By using the UMP container for everything, it inherently supports legacy MIDI messages within the new framework [4]. The ability to plug in a MIDI 1.0 device and still have it operate (with automatic conversion to UMP) is a big plus. It means users don't need separate code paths or libraries for old vs new devices – Teatro handles both seamlessly. This strength is crucial in the transitional period where MIDI 1.0 and 2.0 gear will co-exist for a long time. Notably, Linux's sequencer does this conversion automatically [5]; having similar functionality in Teatro validates that the project is on par with industry best practices for compatibility.

- **Cross-Platform Ambition:** The project's goal to support Linux and macOS (and presumably Windows eventually) is a strength in itself. MIDI 2.0 adoption will be accelerated by solutions that are not siloed to one OS. Teatro providing a unified API or library across platforms means developers can write code once and use it on their DAW or hardware running any OS. This is a competitive edge over something like CoreMIDI (which is Mac-only) or ALSA (Linux-only). It also differentiates from certain manufacturer SDKs that might only target Windows or Mac. By covering multiple OS, Teatro also encourages consistent behavior: for example, a MIDI 2.0 message generated on Linux through Teatro will be formatted the same way on Mac through Teatro, reducing platform-specific bugs. The internal design likely abstracts differences in ALSA vs CoreMIDI, etc., which is a complex task – pulling it off is commendable.

- **Open Source Transparency:** Being on GitHub and open-source (presumably under a permissive license) is a major strength for an implementation of a new standard. It allows the **developer community to audit, contribute to, and improve** the code. This is particularly important with MIDI 2.0 because the spec, while finalized, is still being interpreted and explored by developers. If there are ambiguities or updates, an open project can adapt quickly. It also gives device manufacturers or advanced users a reference they can look at (unlike proprietary OS implementations). For instance, if someone is building their own DIY MIDI 2.0 controller and wonders how to handle property exchange, they could look at Teatro's code for guidance. The openness fosters trust; users can verify compliance rather than taking claims at face value. Moreover, contributions from others (bug fixes, new features like a profile implementation for MPE, etc.) can make Teatro evolve faster than a closed project. This community-driven aspect is a strength relative to, say, waiting for Microsoft or Apple to adjust their closed libraries.

- **Strong Emphasis on Spec Compliance:** Throughout the design, it's clear Teatro's developers were spec-driven. This prevents a lot of problems down the line. For example, by implementing **MIDI-CI** fully, Teatro avoids a scenario where it only works if forced into MIDI 2.0 mode manually; instead it can negotiate properly. Their compliance focus likely means they also handle weird corner cases (like running status doesn't exist in UMP, or how to form valid UTF-8 property data in SysEx, etc.). A spec-focused implementation tends to be **robust** – it won't break when connecting to other compliant devices or software. This interoperability is a huge strength: with MIDI 2.0, everything is new, so ensuring Teatro speaks it correctly means it can talk to hardware from e.g. Roland or Korg that is MIDI 2.0 compliant, with minimal fuss.

- **Holistic Approach (End-to-End):** Teatro seems to handle the whole chain: from **device discovery** (via MIDI-CI) to **message encoding/decoding** to **higher-level feature enablement** (profiles and property exchange). This end-to-end coverage is a strength because it means the library can be used stand-alone to achieve a working MIDI 2.0 setup. One doesn't need to combine multiple tools (e.g., use one library for UMP and another for negotiation). It's all in one place, which simplifies development and reduces potential integration bugs between disparate components.

- **Performance Conscious Design (Potentially):** While we don't have benchmarks, we can infer some performance positives. Using UMP (32-bit aligned messages) is inherently more CPU-friendly on modern architectures than parsing variable-length MIDI 1.0 byte streams. If Teatro is built around 32-bit words, it likely takes advantage of this alignment – for instance, processing messages in bulk or using 32-bit comparisons for statuses. Also, by using native OS mechanisms where possible, it leverages highly optimized system code (like kernel buffering, etc.). If the code is modern C++ and uses move semantics, constexpr for lookup tables, etc., it can be quite efficient. The fact that it deals in raw data when needed means it can achieve low latency (there's not heavy layering if they chose to sometimes operate on raw UMP for speed). We noted earlier the potential pitfall of too much dynamic allocation, but the strength could be if they avoided that – e.g., using stack-based arrays for message assembly. If so, Teatro could handle very high message rates (important for high-res controllers) without choking.

- **Adaptability and Extensibility:** Because it's spec-complete, Teatro is in a good position to adapt to changes or new profiles. If tomorrow the MMA announces a new Profile (say "Guitar Profile" for MIDI guitars), Teatro's infrastructure for profiles can accommodate it by adding profile definitions. The heavy lifting (the SysEx negotiation and enabling/disabling logic) is already done – that's a strength. Similarly, if a **MIDI 2.0 over Network** standard becomes official (there is work on MIDI 2.0 CI over network and UMP over IP), Teatro's modular approach means adding a new transport backend should be feasible without rewriting core logic. This suggests a long lifespan: the project won't become obsolete when new transports or minor spec revisions occur; it can incorporate them.

In short, **Teatro's strengths** lie in its **thoroughness**, **cross-platform reach**, and **community-friendly nature**. It has positioned itself as an early mover in MIDI 2.0 software, which is no small feat given the complexity of the standard. Many projects and developers stand to benefit from the groundwork Teatro has laid in deciphering and implementing MIDI 2.0's intricacies.

Weaknesses and Areas for Improvement

No implementation is perfect, especially with a standard as new and multifaceted as MIDI 2.0. Based on our analysis and what we can surmise about the Teatro codebase, here are some **weaknesses or potential areas for improvement**:

- **Lack of Broad Testing and Maturity:** MIDI 2.0 is still relatively fresh, and Teatro, being an open-source project, may not have gone through the extensive testing that official OS implementations have. There's a risk of **edge-case bugs** – for instance, how does it handle two simultaneous SysEx streams, or unusual message ordering, or devices that only partially implement the spec? If Teatro has primarily been developed in-house or by a small team, it might not have been used in as many diverse scenarios yet. This is a weakness simply due to youth; the project might not yet be battle-hardened. The remedy is more testing – perhaps via automated tests, and by users trying it with various hardware and reporting issues. In contrast, Apple and ALSA implementations are vetted by many engineers and early adopters. To catch up, Teatro should incorporate a comprehensive test suite (if not already present), including using the MIDI 2.0 Workbench as a peer tester.

- **Partial Windows Support (or None Yet):** As noted, Windows is the one platform not explicitly mentioned in the query but is implied in cross-platform ambitions. If as of now Teatro does **not support Windows MIDI 2.0**, that's a gap. It means any cross-platform claims are weakened because one major OS is missing full functionality. This could be a deterrent for adoption in the near term, as many musicians and studios use Windows. Even though Windows' MIDI 2 support was late, it's now becoming available, so Teatro should not lag too far. If currently on Windows it falls back to MIDI 1.0 or is untested, that's an area to prioritize. The improvement would be integrating the Windows MIDI Services as soon as practical. The **weakness** isn't in design (we assume they planned for it), but in execution timeline – needing to catch that wave promptly.

- **Potential Over-reliance on Raw Processing:** From the Workbench's caution and our earlier discussion, if Teatro in its early development took a similar approach (doing everything in raw data without OS integration), this could be a weakness in terms of **maintenance and reliability**. For example, parsing raw USB descriptors to detect MIDI 2.0 devices and managing OS threads for MIDI could be error-prone and duplicative. If Teatro currently bypasses ALSA or CoreMIDI and tries to implement a custom driver or uses outdated APIs, it might suffer from performance issues or compatibility issues. One sign would be if Teatro on macOS is using the old MIDIPacket API to send 2.0 messages (stuffing them into fake SysEx), which might work in limited cases but is not officially supported. That approach could break in subtle ways (e.g., if the OS doesn't expect certain messages via old APIs). Similarly, on Linux, if it doesn't use the `/dev/ump` and instead reads from `/dev/midi` assuming raw bytes, it might never see the 2.0 data properly, or it might not take advantage of kernel timestamping, etc. This is a weakness to address by refactoring the platform backends to use the latest facilities. The project maintainers might need to accept dropping support for older OS versions in favor of doing it the *right* way on new OS versions – which is usually a reasonable trade for a cutting-edge feature like MIDI 2.0.

- **API Usability and Documentation:** While not a "code bug", the usability of Teatro's API could be an issue. If the library is hard to use or poorly documented, that's a weakness for developer adoption. For instance, if enabling MIDI 2.0 on a port requires calling a series of low-level functions in the right order (like manually doing the CI negotiation), some developers might shy away, preferring a library that abstracts that. Ideally, Teatro would provide a simple interface: e.g., `TeatroPort.open(device)` and it internally handles negotiating MIDI 2.0 and setting up

profiles etc., then the developer just sends notes. If currently the developer has to handle those steps manually using Teatro's building blocks, it could lead to misuse or frustration. Also, how errors or statuses are reported is key – if something fails in negotiation, does the library inform the user or just silently fall back to MIDI 1? Clear documentation of these behaviors is needed. If documentation is sparse (common in young projects), improving it is important. This includes not just how to use the API, but also clarifying what *isn't* supported or any quirks. For an internal development perspective, this might be on the roadmap already, but it's worth highlighting: good docs and examples will significantly enhance Teatro's impact.

- **Profile Support Depth:** Implementing the *mechanism* for MIDI 2.0 Profiles is one thing (i.e., enabling/disabling profiles via SysEx), but providing *actual definitions* or behavior for known profiles is another. The MIDI Association has defined a few standard profiles (e.g., MPE, Drawbar Organ Profile, etc.). Does Teatro come with knowledge of these? If not, then after negotiating profile enable, it might be up to the user to interpret the profile data. That could be seen as a gap: for example, if a device says it supports the MPE profile, ideally the library could automatically configure itself (spread channels, etc.) or at least provide helper functions to deal with MPE data. If Teatro doesn't include such conveniences, users might wonder what to do next. Incorporating at least the well-known profiles into the library's logic (perhaps as optional modules) could be an improvement. If those are absent, that's a current weakness in terms of *completeness of the high-level feature*. (This is somewhat speculative; the project may or may not have done this yet.)

- **Property Exchange Support and Complexity:** Similar to profiles, property exchange (PE) could be only partially handled. The mechanism might exist (i.e., sending/receiving the JSON SysEx blobs), but does Teatro actually *parse* and make use of the data? For instance, if a synth sends its device info as a JSON string in PE, does Teatro provide a parsed object to the developer? Or does it just hand them the raw JSON and leave parsing to them? Ideally, a library would parse it and maybe even provide a structured representation or even callbacks for known property schemas. If Teatro currently just moves the data but doesn't interpret it, it's not fully leveraging property exchange. While parsing arbitrary JSON might be out of scope (given extensibility), having at least a JSON parser integrated or examples on how to extract common properties would be good. Without that, the PE feature, though present, might be underutilized by users (because interpreting the spec for each property manually is daunting). This is both a weakness and an opportunity (to build a higher-level API on top of the raw PE mechanism).

- **Memory Footprint:** Depending on how it's implemented, the addition of many MIDI 2.0 features can increase memory usage. For example, maintaining state for profiles and properties might involve storing sizable data (profile definitions, property data caches, etc.). If Teatro is intended for embedded or resource-constrained environments (the query doesn't specify, but if "fully MIDI 2.0 compliant" it might be too heavy for microcontrollers anyway), then memory could be a concern. On a PC, this is less an issue, but still something to watch – e.g., if every MIDI message is wrapped in a heavy object, large MIDI streams could consume a lot of memory quickly. Monitoring and optimizing memory use would be wise, though we can't assert the current state without code. Just noting it as a possible area if the design didn't account for it.

- **Concurrency Issues:** If not designed carefully, MIDI I/O can run into concurrency problems – for instance, receiving data from a background thread while the main thread is sending or changing profiles. A weakness could be if Teatro does not clearly define thread-safety or provide thread-safe mechanisms. The internal feedback aspect of this report might consider if locks are in place or if callbacks can disrupt internal state. Without specific knowledge, we flag it generally: asynchronous MIDI-CI negotiations (which might happen while other MIDI messages are

flowing) can introduce tricky timing; if Teatro's code is single-threaded, it might stall normal message processing during a negotiation. That's not ideal for performance (negotiation could be done in parallel with playing notes if done right). So improving concurrency handling, perhaps by doing negotiation handshakes in a separate thread or state-machine that doesn't block music data, could be an area to work on.

- **Integration with Host Software:** If Teatro is to be used inside DAWs or existing MIDI routing software, how easily does it integrate? For instance, can it create virtual MIDI ports on macOS or Linux so that other software can send MIDI 2.0 data to it? If not, it might be seen as a standalone island. On macOS, creating virtual sources through CoreMIDI is possible – Teatro could expose a virtual MIDI 2.0 port. On Linux, ALSA sequencer can create virtual clients. If Teatro doesn't do this yet, adding that feature would make it more useful (it could act as a MIDI 2.0 router or processor between apps). Without it, it might only handle hardware devices that it explicitly opens, which is somewhat limiting.

In conclusion, while Teatro is fundamentally strong, these **weaknesses** point to refinement needed mostly in **usability, integration, and ensuring alignment with best practices**: - It needs to **embrace OS-level support** fully (to avoid reinventing wheels and possibly suffering performance or compatibility issues) – any lag in doing so is a weakness to correct. - It should focus on **ease of use** (documentation, simpler APIs) so that its powerful features are accessible and not only understandable to the library authors. - It must **broaden support (Windows)** and deepen certain features (profiles/PE) to truly cover "fully MIDI 2.0 compliant" in a way that's also fully *usable*. - And like any young project, more **testing and community feedback** will be crucial to harden it; embracing that will turn current unknowns (possible bugs) into known and fixed issues.

Addressing these areas will enhance Teatro's robustness and appeal, making it a stronger foundation for MIDI 2.0 development.

Recommendations for Improvement

Based on the analysis of Teatro's MIDI 2.0 implementation – covering compliance, code quality, cross-platform integration, and comparisons – the following recommendations are offered to strengthen the project:

1. **Leverage Native OS Support Fully:** Transition Teatro's platform backends to use the native MIDI 2.0 facilities on each OS wherever possible. On **macOS**, utilize CoreMIDI's **MIDIEventList** and related APIs for sending/receiving UMP packets [15] [14]. This ensures compatibility with Apple's implementation and offloads low-level work to the OS. On **Linux**, require ALSA 1.2.10+ and use the sequencer or rawmidi interfaces for UMP [5]. Specifically, use `snd_seq_event_input()` to get UMP events or the new ALSA lib functions (if available) for MIDI2. This will automatically handle bridging between MIDI 1 and 2 clients, as well as any future ALSA improvements, reducing maintenance. For **Windows**, closely track and adopt the **Windows MIDI Services** API as it becomes widely available [18]. Microsoft's new API will be the path forward for MIDI 2.0 on Windows; implementing support for it in Teatro's Windows backend (and deprecating old WinMM usage for 2.0) will complete cross-platform coverage. In summary, **don't re-implement drivers** in user-space that the OS can do better – use the OS as the foundation. This will improve reliability and compatibility, and align with official guidance (even the MIDI 2.0 Workbench advises using OS APIs where available [23]).

2. **Refine and Document the API for Usability:** Review Teatro's API from a developer's point of view and simplify where possible. Provide high-level abstractions for common tasks. For example, implement functions or methods such as `openMidiPort(portName, enableMidi2=true)` that internally handle discovery and protocol negotiation. Offer straightforward methods like `sendNoteOn(channel, note, velocity)` without requiring the user to manually assemble a UMP – the library can do that under the hood based on whether the port is in MIDI 2 or 1 mode. Similarly, for receiving, allow the user to register a callback that provides already-parsed events (e.g., a callback that directly gives NoteOn events with full 32-bit velocity, rather than giving raw packet data). While low-level access should remain available for power users, these conveniences greatly improve **developer experience**. Alongside API refinement, **create thorough documentation** and examples. A README or Wiki should explain how to initialize the library on each OS, how to enumerate ports, how to send messages (with both MIDI 1 and 2 examples), how to use profiles and property exchange. Include a section on migration: e.g., "If you have a MIDI 1.0 app, here's how to upgrade to use Teatro for MIDI 2.0." Well-documented usage patterns will lower the entry barrier and encourage more testing and adoption. In internal development, treat documentation as part of the deliverable, not an afterthought.

3. **Enhance Modular Structure and Code Clarity:** Ensure the codebase is cleanly separated into modules: *Core MIDI 2.0 logic*, *Platform integration*, *Utilities*, etc. If any spaghetti or cross-dependencies exist, refactor to isolate concerns. For instance, a **UMP parser/generator module** should handle encoding/decoding of messages from a byte stream or to a struct, independent of where the data comes from. A **MIDI-CI module** could encapsulate the state machine for discovery, profile negotiation, and property exchange (perhaps with callbacks or virtual functions to retrieve/send SysEx via whichever backend is active). This makes the code easier to maintain and test in pieces. It also allows future extensions (like adding a Network MIDI module) without touching core logic. Where possible, adopt **strong typing** for MIDI data as discussed – e.g., use enums for message types, structs for messages – to avoid magic numbers. This will catch errors at compile time and make the code self-documenting. The comparison with NI's library [9] is apt; consider integrating some of their patterns (they might even accept contributions or you could use their library internally for the data representation). By making the code more declarative (expressing *what* a message is rather than *bit-twiddling* at every use), you'll reduce bugs and make contributions from others easier (because they can understand the structures). In short, invest some effort in **code quality improvements** now that the core functionality is in place – it will pay dividends as the project grows.

4. **Implement Profile-Specific Helpers:** Augment the MIDI-CI Profile support with knowledge of known profiles. For each official profile (MPE, Drawbar Organ, Percussion, etc.), provide an identifier and perhaps a utility class or settings structure. For example, if the **MPE (MIDI Polyphonic Expression)** profile is enabled on a port, Teatro could expose an API like `teatroPort.isMPEEnabled()` or even automatically adjust behavior (like interpret per-note controllers accordingly). At minimum, have constants for profile IDs and maybe a brief description, so that when a profile is negotiated the developer can log "Profile X enabled". For user convenience, consider adding support to automatically **enable certain profiles** if both sides advertise them. E.g., if a device supports MPE and the user flags that they want MPE, Teatro can send the necessary SysEx to turn it on. These helpers turn raw profile support into actual practical use of profiles. Similarly, extend **Property Exchange** handling: if feasible, integrate a lightweight JSON parser (there are many MIT-licensed single-header JSON libraries for C++). Use it to parse incoming PE data into a dictionary or object, so the developer can query properties in a structured way (e.g., `deviceInfo = port.getProperty("DeviceDetails")` returning a structured object with manufacturer, product, etc.). For outgoing, maybe provide a

builder for property exchange requests. This would significantly enhance the usability of profiles and PE – features that are otherwise quite complex to use correctly. Essentially, move up one level of abstraction from just "transporting SysEx" to "manipulating high-level profile and property data".

5. **Performance Tuning and Testing:** Do a pass to identify any performance bottlenecks or inefficiencies in the MIDI 2.0 data path. This includes:

6. Checking if message parsing and construction can be done with fewer temporary objects (e.g., perhaps reuse a single UMP buffer for all outgoing messages on a port, rather than allocate each time).
7. Ensuring that heavy operations (like parsing a large JSON SysEx for property exchange) do not occur on real-time threads. Those should be offloaded to worker threads so as not to stifle the flow of musical messages.
8. Utilizing lock-free queues or double-buffering for MIDI input if multiple threads are involved, to avoid mutex contention at high event rates.
9. If using C++, enable optimizations like `constexpr` for lookups (for example, you might create compile-time tables for mapping status byte to message length, etc., so it's O(1) with no branches at runtime).
10. Evaluate the memory usage when, say, sustaining 1000 messages per second; profile the code with tools to ensure no unusual spikes or leaks.
11. Since jitter is critical in MIDI, verify that Teatro introduces minimal scheduling jitter. This can be tested by timestamping outgoing messages and comparing to a reference (if the OS provides high-res timestamps for when messages actually went out, e.g., CoreMIDI does so). If any part of the code adds unpredictable delay (like waiting for a negotiation mid-performance), restructure to avoid that.
12. For throughput, maybe stress test with an "all controllers moving" scenario or a dense MIDI file, to see if any code path (like debug logging in the middle of processing) slows it down.

The outcome of this analysis might suggest micro-optimizations or larger design tweaks (for instance, if you find that converting every MIDI 1 message to UMP individually is slow, maybe batch convert or keep an internal cache of common conversions). Given that ALSA and CoreMIDI handle a lot efficiently in kernel/C, bridging into them should keep overhead low, but the user-space parts (CI, profile management) should be similarly scrutinized.

1. **Robust Testing and Validation:** Increase automated testing of the MIDI 2.0 implementation. Create unit tests for UMP encoding/decoding – e.g., given a known MIDI 2.0 Note On structure, does it produce the correct 4 bytes, and vice versa? Test edge conditions like Note Off with velocity, RPN messages with min/max values, etc. Set up integration tests if possible: perhaps a virtual loopback where Teatro sends messages that Teatro then receives, verifying nothing is lost or altered. Use the **MIDI 2.0 Workbench** or similar tools to validate compliance: for example, use Workbench to send various CI scenarios (discovery request/response) to Teatro and ensure Teatro responds appropriately. If any compliance test fails, fix it. Additionally, test with real devices if available – e.g., Roland's A-88MKII keyboard (now MIDI 2.0 via update), or the new class-compliant MIDI 2.0 devices, to ensure Teatro can negotiate and communicate with them. Incorporate regression tests for any bugs found so they don't recur.

Also, consider continuous integration with different OS targets: e.g., set up CI runners for Linux and macOS that run the test suite. This will catch platform-specific issues (like if something works on Linux

ALSA but fails on Mac, CI can flag it). With the complexity of multiple protocols and platforms, automated testing is your friend.

1. **Engage with the MIDI 2.0 Developer Community:** Since this feedback is internal, it's worth planning outward engagement once improvements are made. For instance, consider contributing to or referencing **midi2.dev** resources. If Teatro becomes stable and well-documented, you might propose it be listed on midi2.dev as an available open-source library. This will attract more users and contributors. Engage on forums like the MIDI Association forum or the JUCE forum's MIDI 2.0 threads to share experiences. Sometimes, others have solved similar problems (for example, the developer of another library might share how they handled running status conversion, etc.). Collaborative spirit is high in the MIDI 2.0 rollout phase. Also, keep an eye on updates from the MMA – if new versions (UMP 1.2 or MIDI-CI 1.3 etc.) appear, plan to update Teatro accordingly. Early adoption of spec updates can be a selling point. By staying plugged in, the team can anticipate changes and gather feedback on desired features.

2. **Expand Cross-Platform Scope When Ready:** Once the core is strong on desktop OSes, consider if there's demand for other platforms:

3. **Mobile (Android/iOS):** Android doesn't yet support MIDI 2.0, but if it does, being prepared (Teatro for Android via Oboe or AAudio maybe) could be forward-thinking. iOS likely will mirror macOS's CoreMIDI, so supporting iOS is mostly a matter of build configuration if using CoreMIDI APIs.

4. **WebAssembly/WebMIDI:** It's a bit early since WebMIDI doesn't have MIDI 2.0, but Web serial or WebUSB might allow connecting to MIDI 2.0 devices from the browser. A pared-down version of Teatro compiled to WebAssembly could serve web applications in the future. This is speculative, but it's how far-reaching a solid implementation can go. These aren't immediate needs, but as an internal vision, keep the code flexible enough (and free of OS-specific assumptions) so that porting is feasible.

5. **Provide Fallbacks and Graceful Degradation:** Ensure that when connected to a MIDI 1.0-only environment, Teatro doesn't break or produce errors – it should smoothly operate in MIDI 1.0 mode. For example, if a device or OS doesn't support property exchange, Teatro should detect that and simply not attempt it, rather than hang waiting for a response. Graceful handling of "not supported" responses in MIDI-CI is key (the spec defines how a device can NAK a request). Teatro should handle those cleanly and perhaps notify the user which features are unavailable. This makes the library robust in mixed setups. It's worth adding tests for such scenarios (simulate a device that says "no profiles support" or "only MIDI 1.0 protocol supported" in discovery). The library should then automatically use MIDI 1.0 messaging for that port (which it can still encapsulate in UMP for internal uniformity). Documenting this behavior (so developers know what to expect if, say, their device isn't actually MIDI 2.0 capable) will avoid confusion.

6. **Memory and Resource Management:** Audit the library for any resource leaks or excessive usage. Ensure that ports are properly closed and memory freed when a port or the library is shut down. If Teatro creates any background threads (for listening to MIDI input, etc.), make sure they clean up and don't prevent application exit. On Linux, check that sequencer clients are destroyed when done, so they don't accumulate ghost clients. On Mac, ensure MIDI client/port references are released appropriately (CoreMIDI objects). Also, handle hot-plug events (devices connecting/disconnecting) if applicable – the library should either notify the user or handle it seamlessly. For example, if a MIDI 2 device is unplugged, any waiting property exchange should abort gracefully.

> These are more about resource robustness than core functionality, but important for integrating into real-world applications that run for long periods or dynamically change configurations.

By implementing these recommendations, **Teatro** will significantly strengthen its MIDI 2.0 offering. To summarize, the key themes are *align with OS capabilities*, *make the library easy and safe to use*, *cover all bases with thorough testing*, and *be ready for the future*. The work done so far is a strong foundation – these steps will ensure that foundation supports a broad and lasting structure, establishing Teatro as a reliable cornerstone in the MIDI 2.0 ecosystem for developers and musicians alike.

---

1 2 3 4 5 11 MIDI 2.0 on Linux — The Linux Kernel documentation
https://docs.kernel.org/sound/designs/midi-2.0.html

6 Details about MIDI 2.0 | Hacker News
https://news.ycombinator.com/item?id=20007638

7 8 9 28 MIDI2.dev: Open-Source Resources for MIDI 2.0 Developers – MIDI.org
https://midi.org/midi2-dev-open-source-resources-for-midi-2-0-developers

10 23 30 31 GitHub - midi2-dev/MIDI2.0Workbench: MIDI 2.0 Testing Tool
https://github.com/midi2-dev/MIDI2.0Workbench

12 Detailed changes v1.2.12 v1.2.13 - AlsaProject
https://www.alsa-project.org/wiki/Detailed_changes_v1.2.12_v1.2.13

13 And if Zynthian was the first "hardware" synthesizer supporting MIDI ...
https://discourse.zynthian.org/t/and-if-zynthian-was-the-first-hardware-synthesizer-supporting-midi-2-0-ump/7353

14 CoreMIDI sample using new API for MIDI2 - GitHub Gist
https://gist.github.com/sonsongithub/6520ac3f537e3e9b7d3f74e576ae70c5

15 Incorporating MIDI 2 into your apps | Apple Developer Documentation
https://developer.apple.com/documentation/coremidi/incorporating-midi-2-into-your-apps

16 17 MIDI 2.0 Support (MIDI-CI/UMP) · Issue #7 · orchetect/MIDIKit - GitHub
https://github.com/orchetect/MIDIKit/issues/7

18 Windows MIDI Services October 2024 Update
https://devblogs.microsoft.com/windows-music-dev/windows-midi-services-oct-2024-update/

19 22 Microsoft's Major Moves To Make Making Music on Windows Easier
https://midi.org/microsofts-major-moves-to-make-making-music-on-windows-easier

20 Status of MIDI 2.0 in JUCE 8 and ETA for Windows support etc
https://forum.juce.com/t/status-of-midi-2-0-in-juce-8-and-eta-for-windows-support-etc/61190

21 25 JUCE Roadmap Update Q3 2024
https://juce.com/blog/juce-roadmap-update-q3-2024/

24 MIDI 2.0 Preview Branch - News and Announcements - JUCE Forum
https://forum.juce.com/t/midi-2-0-preview-branch/66453

26 Windows 11 Insider Canary build 27788.1000 - Feb. 5
https://www.elevenforum.com/t/windows-11-insider-canary-build-27788-1000-feb-5.33086/

27 Microsoft debuts Windows MIDI Services with MIDI 2.0 support
https://www.theregister.com/2025/02/06/windows_midi_services_2/

29 MIDI2.dev · GitHub
https://github.com/midi2-dev