Synchronisation and Orchestration of Mechatronic Instruments

Anfri Hayward

***Abstract*—VUW has developed three innovative mechatronic instruments capable of complex, human-like playback. However, no system currently exists to orchestrate these instruments as a singular ensemble. Each instrument features unique latency and instrumentation characteristics that must be addressed for synchronous playback. This paper presents MechSync, a software-based scheduling and distribution system designed to synchronize the playback of a diverse range of mechatronic instruments. By facilitating communication over a unified MIDI protocol, MechSync allows instruments from various creators to be played together, with each system's characteristics managed by specialized modules.**

***Index Terms*—Software Engineering, Mechatronic instruments, Synchronisation, Scheduling, Asynchronous pipelines, MIDI communication.**

I. Introduction

VUW has previously developed three mechatronic instruments: MechBass, DrumBot, and Azure Talos. These instruments are highly specialised, and are capable of expressive playback, utilising diverse techniques. However, interfacing with these instruments poses unique challenges. Each instrument has been developed by different students over multiple years, with diverse latency and performance characteristics. Whilst effective at individual performance, the bespoke interfacing of each instrument makes synchronised playback as an ensemble challenging. This necessitates the need for a unified intermediary framework for interfacing. VUW's mechatronic instruments are often toured for exhibitions, showcasing the university's innovations in engineering and design. Thus by developing this system, we can further highlight the capabilities of VUW's unique mechatronic instruments, and contribute to the broader landscape of music technology.

The resulting artifact of this project is MechSync, a dynamic framework for the synchronisation of mechatronic instruments. MechSync has been designed as a graph-based temporal compensation framework, utilising the MIDI protocol for communication. The software processes MIDI instructions in real-time, utilising specialised graph nodes to handle the individual characteristics of each instrument integrated within the ensemble.

The initial scope for this project sought to create a framework which could synchronise a single string of MechBass with a single drum of DrumBot. However, the final artifact of MechSync has far exceeded this metric, and is able to synchronise all aspects of playback between both instruments. Additionally, the framework has been built to be fully dynamic, such that it can be expanded via plug-ins to introduce new instruments as they are developed. MechSync can also be configured with arbitrary instrument ensembles using configuration files, further enabling flexible playback. This adaptive structure ensures that MechSync can be sustainably maintained into the future as new mechatronic instruments are developed.

This project was supervised by Dale Carnegie and Jim Murphy.

A. Specifications

To prove the effectiveness of MechSync, both metric-based and human-based evaluation techniques have been outlined. Effective synchronisation requires that any residual latencies present in playback are below the tolerances of human hearing. Utilising the Random Gap Detection Test, Owens et al. found that the minimum discernible latency was 9.23 – 12.15 ms [1]. Thus, the maximum desired latency of MechSync was determined to be 10 ms. To evaluate the framework against this metric, a testing suite was produced which sends concurrent note instructions to multiple instruments simultaneously. The playback of each instrument is recorded and evaluated to detect amplitude peaks within the audio waveform. Spikes within the audio signify the impulse moment of each instrument. From this, the spikes within the recordings can be mapped to notes within the reference track to evaluate the temporal offset between instruments. This evaluation found that MechSync was fully capable of synchronising the average latency within the mechatronic ensemble to within the 10 ms tolerance required for synchronous playback.

Additionally, MechSync has been evaluated by members of the public. MechSync was showcased during the 2024 VUW Open Day, and thus was able to be evaluated within a public context. Feedback from this event was highly positive, with viewers impressed with the capabilities of MechSync to fully utilise the playback features of MechBass, which was utilised for the showcase. The latency and playback quality of MechSync has also been evaluated by the author of the framework, who is a musician with 8 years experience playing percussion instruments.

B. Structure

This report seeks to outline the processes undergone during the development of the MechSync framework. Related fields and projects will be outlined, showcasing existing work within the synchronisation space, and how these works have inspired and informed the development of MechSync. Additionally, this section will highlight the libraries and frameworks considered, and justify the choices of tools for the final artifact. This is followed by a thorough description of the underlying design and implementation of MechSync, discussing the graph-based architecture in detail. Following this, the evaluation and findings of the software will be covered, showcasing the efficacy of the framework with orchestration and synchronisation. Finally, the existing limitations of MechSync will be discussed, with an overview of the future works enabled by the framework.

II. Related Works

A. Synchronisation

Chronological synchronisation continues to be a challenge within modern disconnected systems. Within a system such as a mechatronic musical ensemble, ensuring that disjoint actors perform synchronously is paramount. Each actor within an ensemble possesses its own latency characteristics, and so these must be accounted for when orchestrating actions. In addition, actors themselves may possess some degree of internal latency, which affects the actions that may be performed within a given state. For example, if DrumBot is requested to play two disjoint drums in quick succession, it might be unable to perform the action due to panning latency. Therefore, the proposed solution must also take into account unperformable actions when scheduling instructions.

When communicating over networks such as the internet, there is an inherent communication delay between when data is sent and received. As separate connections have their own latencies, dispatched data sent over these connections will be received asynchronously, and thus the receiving actors will perform asynchronously. One solution to this problem is the use of dynamic local lag. In [2], Sithu et al. propose a system utilising bi-directional local lag to artificially synchronise two actors over a fixed-latency network. By buffering locally produced data by the latency of the network connection, synchronisation of playback can be achieved. This however, results in a noticeable delay for local playback, reducing the interactivity of the model. Despite this, Sithu et al. found that the use of their dynamic latency buffer improved the mean opinion of users who trialled their system when compared to absent buffering.

Park and Choi [3] further discuss buffering methodologies for synchronous playback. In contrast to [2], Park and Choi primarily focus on mono-directional synchronisation in regards to streaming data over a network. As a stream is transmitted, data packets may arrive with variable latency. To allow for continuous playback, Park and Choi discuss the use of temporal distortion, dynamically adjusting the rate of buffered data playback. Park and Choi further extend this concept via inter-media synchronisation. Temporal distortion is utilised between disjoint streams, allowing for synchronisation of media from multiple sources with unique latency characteristics. This results in a consistent and synchronised playback, even when streaming media with highly varied latency characteristics.

The firmware of MechBass features no synchronisation functionality. This means that whilst MechBass immediately responds to received instructions, the variable duration required to pan to the correct bass fret results in jumbled playback. To resolve this, MechSync utilises variable delays to adjust the times when data is transmitted to the receiving instrument. Unlike [3], the temporal characteristics of mechatronic instruments such as MechBass are consistent given

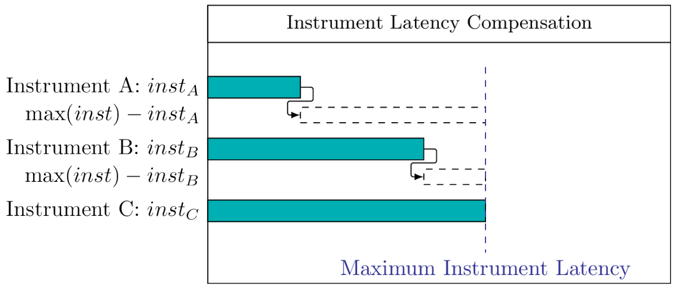

Fig. 1. Example of latency compensation.

a known transition in state, such as panning between two notes. This means that latencies can be precomputed and applied prior to transmission, allowing for total decoupling between the instrument's firmware and latency compensation mechanisms.

Zhao et al. [4] describe an alternative model for handling latency within distributed systems. In contrast to [2], [3], Zhao et al. propose the use of timestamped instructions, which are buffered locally by each actor. Internal clocks synchronised with the orchestrator are used to ensure that each actor performs the given instruction simultaneously. The earliest time that a given instruction can be executed is the maximum time required for all actors to receive the instruction.

Despite proving effective for the orchestration of host-actor distributed systems, it is not appropriate for the application of mechatronic instrument synchronisation. Firstly, timestamp synchronisation requires that each instrument takes an active role in synchronisation. This would require every instrument to have its own latency compensation code and internal clock, which may be infeasible on the existing instrument hardware. Additionally, each mechatronic instrument communicates via MIDI. As MIDI is a bufferless, instantaneous protocol, it does not implement timestamp functionality. This means that there would be no feasible method for transmitting timing information without breaking MIDI compatibility, limiting the ability for the instruments to perform individually.

Based on the works of [2], [3], MechSync uses a hybrid synchronisation scheme featuring both temporal distortion and dynamic buffering. Some mechatronic instruments feature their own scheduling protocols which introduce a fixed latency for communication. DrumBot features a fixed 1970 ms buffer which is used to account for the panning time of its arms. MechSync accounts for this latency by buffering data to an instrument by the difference of the maximum latency of the network (See Figure 1). This ensures that instruments with greater latencies receive their instructions earlier to compensate.

B. MIDI Processing

As MIDI is the primary communication method for both DrumBot and MechBass, it is important that the method of MIDI processing and communication is robust and flexible. A key constraint is the method in which MIDI data is ingested by the software. One possible choice is the use of MIDI files. A wide range of MIDI file processing libraries exist for many languages. PortSMF [5] is one such library for C++, operating as a component of the PortMedia project. PortSMF is also used by the Audacity project [6] for handling MIDI file input. Java features the `javax.sound.midi` [7] API, which allows for both MIDI file processing and MIDI output to physical devices. This API is also available within Kotlin, which is why this language was initially selected in the project proposal.

Unfortunately, MIDI files impose some heavy restrictions on usage. Playback is limited to pre-constructed files, and integration with live systems such as DAWs (Digital Audio Workstations) is not possible. Providing support for MIDI files also requires supporting standard mappings between instruments and outputs [8], and adjustable playback, further complicating the implementation of the final artefact.

An alternative approach is the use of virtual MIDI ports. Virtual ports masquerade as physical MIDI devices but forward their data to a bound program. This allows standard software such as DAWs to interface with them as if they were physical MIDI hardware. If MIDI file playback is desired, many programs exist which can pipe files directly into a port, such as ALSA's pmidi [9], which comes prepackaged with most Linux distributions. On Windows platforms, an SDK [10] has been developed to allow for the dynamic creation of virtual MIDI ports. Unfortunately, as this project targets Linux-based systems, this library is not appropriate. Additionally, the licensing of this SDK is heavily restrictive, requiring the explicit consent of the author prior to usage. Using this library would therefore have introduced vagaries regarding future development and distribution of this project.

In contrast, the open-source library RtMidi [11] provides an all-in-one MIDI solution, including support for virtual MIDI ports. Whilst this is highly desirable, the API is built primarily for C++ development. This would have heavily increased the workload for implementation, especially as C++ makes implementing many useful synchronisation techniques difficult. Fortunately, RtMidi inspired the creation of Midir [12], a Rust library built with a similar API and equivalent feature-set.

As the Midir library provides a strong backbone for developing with real-time MIDI data, MechSync has been written in Rust, utilising Midir for handling the MIDI interfacing between bound sources and output instruments. Additionally, Rust provides many synchronisation primitives, which aid in the creation of multi-threaded and asynchronous programs such as MechSync.

C. Asynchronous Architectures

When implementing a system featuring scheduled delays, it is very difficult to implement these without utilising some form of asynchronous programming practices. Whilst trivial to write, a sleep call within a thread in a synchronous program is a blocking operation. If we were to loop over this call, this would result in a blocking delay for each iteration. A common approach to handling this is to utilise a buffered queue and looping poll. Whilst easy to implement, this approach is CPU intensive, and requires polling the queue constantly to check for any elements which have waited a sufficient delay. A naive developer may try to utilise threading in order to offload each blocking delay, but this too has implications.

Classical models of threading often result in code which is difficult to reason with. A study by Rossbach et al. investigated the correctness and ease of use of different synchronisation methods within multi-threaded programs [13]. One such method investigated was fine-grain locking, which involves implementing mutex locks around key components of a program, seeking to minimise the scopes of these locks. Mutex locks are synchronisation primitives that are often used within multi-threaded programs to prevent multiple threads executing a section of code at the same time. Of the 237 students tested, over 70% of the programs utilising fine-grained locking contained errors. The study did identify that transactional programming using atomic scopes was much easier to reason with, however this model maps poorly onto the requirements for this project. Transactional programming would require that all MIDI data elements processed within the framework occur sequentially. Consequentially, it would not be possible to delay instructions without blocking the pipeline, as atomic scopes would hold the mutex lock throughout the delay.

Traditional OS threads, as used by Rust, come with computational overhead for their creation and destruction. In addition, the performance of threads falls once the hardware threads of a given architecture are saturated. In a situation where we have 100s of notes buffered, a dedicated OS thread for each would produce an excessive overhead.

If we return to the problem of scheduled delays, we can see the underlying issue. We want to be able to pause some task and return to it after a given delay. Within the traditional thread model, we've been sleeping the thread and letting it do nothing. However, it would be more advantageous if we use this thread to process other tasks whilst we wait. This is the core principle behind cooperative asynchronous programming.

One of the leading methods of asynchronous programming is async, with the async/await syntax. By sugaring functions with the async keyword, we can mark that the given function returns some "future", which can be awaited on for the given output. This allows for separation between function calling and the result ingestion. In addition, async allows for the creation of tasks, which can call asynchronous code in parallel. Under the hood, this operates using cooperative multitasking [14]. At each async/await point, the executor can choose to switch to another task. When using specific async sleep functions, this can allow for other tasks to be performed during the sleep period. Rust comes with native support for async/await syntax, although execution specifics are left to library implementations. One of the most prominent async libraries is Tokio [15].

Whilst Rust appears to provide powerful tools for asynchronous programming with async/await, it is decorated in a thick layer of "foot-guns" [16], quirks of implementation which make correct implementation challenging. Rust's async implementation is designed for performance and embedded systems, but this comes at the cost of flexibility. Due to Rust's constraints on lifetime, and because async functions do not know how long they'll live, every parameter to an async function must be lifetime `'static`. Additionally, when an async function is written, the compiler converts it into a function returning an unnameable compiler-defined future type. As a consequence, it is extremely difficult to pass this type around dynamically as it does not have a type which can be named. This becomes an even greater problem when handling async functions as pointers, as the function itself also has an unnameable type. To resolve this, we must build conversion logic to create a dynamic version of this type. The resulting type from this conversion is:

```
Box<dyn Fn(T, ...) -> Pin<Box<dyn
Future<Output = U> + Send + 'static>> +
Send + Sync>>
```

In contrast, a synchronous function pointer type is:

```
Box<dyn Fn(T, ...) -> U>
```

Nystrom further highlights the implications of async with the metaphor of "coloured" functions [17]. All async functions must either be called from another async function, or as part of an async task. Furthermore, async functions must have their results explicitly awaited. This means that semantically, we must treat async functions as a different "colour" of function. As can be seen with Rust, this has widespread implications for the structure of our code. However, Nystrom proposes an alternative solution which shows promise: green-threads.

Green-threads, colloquially known as Goroutines [18], are lightweight thread-like abstractions. Green-threads operate as an implementation of M:N threading, where M user threads are mapped onto N kernel threads [19]. This allows for task-switching to be handled at the user level rather than the kernel level, which is much more performant. Under the hood, Tokio's multi-threaded async handler also uses M:N mapping, running M async tasks on N kernel threads. Many languages natively support green-threads, such as Go's Goroutines [18] and Kotlin's Coroutine [20]. Rust's threading model used to operate on green-threads, but as the language transitioned towards a systems language, the costs of green-threads became an implementation burden [21]. Rust therefore only implements native support for the 1:1 OS thread model.

Fortunately, green-thread support can be re-implemented into Rust using the May library [22]. In contrast to async, May requires minimal syntax to spin up a closure as a new green-thread. In addition, May is compatible with existing functions and types, and avoids the colouring problems of async. Similar to Tokio, May also has specialised synchronisation primitives such as mutex and sleep, which act as non-blocking task switches. May is also highly performant, and is used as the backing for may-minihttp, which is currently the second fastest HTTP API available [23].

Due to its minimal API overhead and implementation flexibility, MechSync utilises the May library for handling the asynchronous processing of MIDI instructions within its processing pipelines. Whilst async is more idiomatic to Rust, with the API built into the language's syntax, green-threads provide a clearer abstraction with less mental overhead required for effective implementation.

D. *Functional Flow*

Structuring a program's flow is important for ensuring appropriate abstractions for the actions to be performed. Within the context of this project, some number of MIDI inputs are expected to be connected to some number of MIDI outputs. For example, Azure Talos functions as six independent modules. Due to this, it is reasonable to assume that Azure Talos will have to interface with six MIDI output devices. To allow for this flexibility, a modular structure is required.

Flores et al. propose the use of pipeline trees for representing branching data flows [24]. By utilising a branching node-graph, it becomes possible to represent a complex branching function as a chain of modular components. In addition, linked nodes allow for arbitrarily constructed pipelines, and the reuse of components. Furthermore, node-based structures can benefit highly from multi-threaded architectures and parallelism. Lee and Messerschmitt highlight the benefits of operating nodes on individual threads [25], as an implementation of horizontal slicing. This architecture is beneficial for systems where different rates of processing may interface in a pipeline, or where tasks are CPU bound. Unfortunately, pipeline processing of MIDI data is primarily an I/O bound task, so this methodology does not provide any benefits. Instead, MechBass utilises vertical slicing, where each data element is processed on a dedicated green-thread. This allows for transforms within a pipeline to be written in a traditional synchronous style, and forgoes the requirement of FIFO primitives between pipeline nodes. Additionally, underutilised nodes within the graph do not incur the cost of spin-locking whilst awaiting input. This chosen architecture is similar to the underlying mechanics of Java's ParallelStream implementation [26].

III. Design

A. *Node Graph*

MechSync utilises a chained call-graph structure to bind virtual MIDI inputs to virtual MIDI outputs. Each node within the graph contains call bindings to some number of child nodes. When MIDI data is ingested by a node, it will perform its logical and temporal transformations to the data, and pass the resulting data to its child nodes. Naturally, this results in a flow of MIDI data from suppliers (inputs) to consumers (outputs).

The temporality of instructions is represented by the time of execution within the pipeline. Instead of accumulating delays within the pipeline and applying them upon output, delays are applied in real-time to data atoms as they are processed. This was chosen as it ensures that all internal states are accurate to the time in which data is processed. This is important, as it is possible for an atom within a pipeline to overtake another atom which was ingested earlier. For example, atom A is received 200 ms earlier than atom B, but atom A is delayed by 400 ms whilst atom B is not. In this case, if we were to utilise accumulated temporal shifting, downstream nodes would still receive A before B. Consequentially, if said downstream nodes were stateful, their state would not represent the actual state of playback; It would be impossible for the system to perform operations based on previous state.

The node graph call structure is implemented using recursive calls to child node consumer functions. The primary motivation for this design was the minimisation of implementation complexity, especially in the event of multiple bound child nodes. A simple example of this would be an echo effect, where any ingested MIDI instruction is echoed multiple times to a given output, with decaying velocity. This requires multiple calls to child nodes with a temporal shift between them. With recursive calls, this is easy to implement, as chained calls can be injected anywhere within the node's execution, dispatched asynchronously as required. Without recursion, we could utilise return chaining via a helper dispatcher, however, this breaks temporality. A return operation represents stack unwinding, and thus must be the final operation within a given stack-frame. Consequentially, we cannot return elements from a given function at different times.

Some programming languages feature generator constructs, which allow for imperative code with multiple returns, wrapping this behaviour as a lazily-evaluated iterator. Unfortunately, whilst Rust implements this behaviour for converting asynchronous code into generators, the generator API based on this underlying implementation is highly unstable [27], [28]. From this, it was decided to use recursive call-chaining.

A consequence of recursion is that every chained call results in a new stack-frame. In environments with limited stack memory, this can limit the depth of traversal allowed within the call graph. Fortunately, MIDI calls within MechSync feature no back-propagation of data, that is to say, each call returns `Unit`, a type with no information. This means that if the last operation within a node is a call to its child, we can discard its stack-frame via tail-call optimisation. The Rust compiler is able to perform this automatically, and thus, we can minimise the memory impact of recursion. However, even without tail-call optimisation, it is highly unlikely that stack-depth will be exceeded. This is because typical usage patterns of MechSync are unlikely to produce a call-chain deep enough to exceed the stack space allocated.

To expand the flexibility of MechSync, node graphs are computed and constructed at run-time via loaded configuration files. This allows for the dynamic creation of call-graphs as required by the user. From this, it is possible to load different ensembles via different graph configurations, minimising wasted resources on unused graph elements, and greatly improving the software's flexibility. By allowing for dynamically created node elements, MechSync enables fully dynamic graph creation, which could be utilised in the future to build real-time graph editing via a UI.

B. Asynchronous Processing

The synchronisation of multiple instruments with unique timing constraints produces many interesting design challenges. As MIDI data is ingested, it must be possible for data to flow through the system at different rates to compensate for the latencies of the mechatronic robots. MechSync achieves this by dispatching each received MIDI instruction to a dedicated green-thread. These threads can be independently paused and resumed via sleep calls, which allows each element to be independently buffered as required. The underlying thread scheduler is not exposed to the developer, and thus allows for imperative single-threaded code with underlying cooperative multithreading.

An additional benefit of this architecture is that it allows for easy forking of data flows by spinning up helper green-threads. These threads are then automatically processed asynchronously with the originator thread, allowing each thread to possess its own timing constraints. Unfortunately, it is difficult to merge these threads together again as they do not return state. Fortunately, there are no parts of MechSync that require this behaviour, and thus the impacts are minimal.

C. MechBass

Communication with MechBass is built around simple MIDI channel dispatch. Each module of MechBass is responsible for a given string, and listens to a unique MIDI channel. When a MIDI instruction is received on its channel, it checks if the note is within the tuning range of the string. If so, the module immediately makes efforts to play the note, panning to the correct fret as required. Unfortunately, this simplified firmware makes synchronised playback difficult. Any users of MechBass must write music which manually specifies the channels for each note, and notes must be temporally offset to account for the latencies of panning between frets. To resolve this, MechSync utilises a custom graph node for handling these intricacies, such that it can play in an arbitrary and synchronous fashion.

To perform state-based decisions, a local copy of MechBass's state is emulated within the MechBass node. This state encodes each module's current note, whether it's playing, the note delay, and the timestamp of when it was last modified. By default, these are initialised as the root position on each string of MechBass. When MechBass is initially powered on, the fret clamps calibrate themselves by shifting towards the root position. From this, MechBass is able remember where each fret clamp is by the rotational offset from this root position. Unfortunately, MechBass is unable to communicate this state to MechSync, as its MIDI communication operates mono-directionally. It is therefore best for MechSync to assume that the state of MechBass is that of root position. If MechBass is not in root position, the first note sent to each module will assume that MechBass is in a different state, and so may be played at the incorrect time. In practice this has little effect, as afterwards, the state becomes known and synchronous playback resumes.

By knowing the previous state of MechBass, it becomes possible to track how far MechBass will need to pan in order to reach a given note. Humans process pitch intervals based on the ratios between frequencies. For example, an octave represents a frequency ratio of 2:1. This means that the change in frequency for intervals is multiplicative. As the frequency of a string is proportional to $\frac{1}{Length}$, this means that distance travelled between frets is non-linear to the interval travelled. Like a typical bass guitar, MechBass is tuned using 12-note equal-temperament. This means that there are 12 notes within an octave, and each note is spaced with the same multiplicative factor relative to its neighbours of $2^{\frac{1}{12}}$. This factor ensures that when multiplication is applied 12 times, we get $2^{12 \cdot \frac{1}{12}} n \to 2n$. Therefore, we can conclude that the length of a string n notes higher than the root must have a length of $\frac{1}{Length} \to \frac{1}{2^{\frac{n}{12}}} \to 2^{-\frac{n}{12}}$. Furthermore, this means that the difference in length of string between any two notes is $|2^{-\frac{n_1}{12}} - 2^{-\frac{n_2}{12}}|$. As MechBass shortens the length of the resonant string by using its fret clamps, this also represents the distance which the fret clamps must pan between 2 notes.

Assuming that the panning motors of MechBass have consistent acceleration profiles, it is possible to map this distance travelled to an expected time of panning. Typically, the movement characteristics of a motor over a given time can be represented by some form of:

$$f(t) = at + bt^2 + ct^3 + ...$$

However, in the case of MechSync, as we are trying to solve for t, we must instead calculate the inverse $f^-1(t)$. As the degree of $f(t)$ is uncertain, and the constants unknown, it is more appropriate to find an approximate representation of the curve. Given some function $f(x) = ax^b$,
$f^-1(x) = a^{-\frac{1}{b}} x^{\frac{1}{b}}$, assuming positive values of x. As a and b are arbitrary constants, we can replace them and simplify to get $f^-1(x) = cx^d$. This means that a basic inverse polynomial can be represented using this exponential equation. By collecting samples for distances travelled by MechBass and their times, it is possible to utilise regression curve-fitting to find a set of coefficients which approximate the behaviour of the hardware with high accuracy. The final approximation curve chosen was $f^-1(x) = ax^b + cx^2$. The additional quadratic component was added to flatten the curve for longer distances, as acceleration decreases as the motors reach full speed, and thus take on a linear component.

By utilizing these derived equations, it becomes possible to predict the time required to pan between 2 notes. To perform synchronous playback, we buffer each note's transmission by $t_{max} - time(distance(n))$. t_{max} can be calculated as a constant of $t_{max} - time(distance(n_{max}))$, where n_{max} is the number of frets on a string. In the case of MechBass, all four strings possess 13 frets, so this constant can be applied to all string modules. By utilizing these equations, we can guarantee that notes will be played in a temporally consistent fashion.

The synchronization of MechBass also requires handling string dispatch. This ensures that notes are sent to their appropriate strings, and allows for context-aware polyphonic playback. When a note-on instruction is received, the state of MechBass is inspected to locate a free channel to play the given note. As the ranges of strings overlap, there may be multiple strings which can play the note, and so a heuristic is used to prioritise the strings.

Initially, channels were prioritised with first-come-first-serve, where notes would be sent to the highest string possible. Ideally, this method ensures that fret clamps are positioned as close to the root of their string as possible, which minimises the travel distance for panning. Unfortunately, channel roll-over resulted in lower strings performing excessive panning, which placed heavy strain on the fret clamp mechanism, and resulted in frequent slippage. To resolve this, channels are instead sorted by the minimum distance between the desired note and the channel's previously played note. This is done to minimise the amount of time spent panning between notes, which reduces mechanical noise, and helps to reduce wear, prolonging the lifespan of MechBass.

From these prioritised channels, the framework will then choose the first free channel available. If there are no channels available, MechSync will override the first available channel, and issue a warning to the console. Whilst this can result in notes being cut short, this is preferable to skipping these notes entirely.

When a note-off instruction is received, the currently playing notes are traversed to locate a note with a matching pitch. The found note is marked as not playing, and the timestamp when it was stopped is recorded. This timestamp allows the string dispatcher to make predictive decisions based on the release time of the string. If the time until MechBass releases a string is shorter than the panning compensation for the next note, the dispatcher is able to choose this string, even if the hardware has not finished playing the note. This ensures that MechSync is capable of fully utilising the capacities of the available string modules.

D. DrumBot

Compared to the MechBass node, the implementation required for DrumBot is comparatively minimal. This is because DrumBot already features its own internal synchronisation scheme and latency calibration system. Each MIDI instruction sent to DrumBot will be delayed for 1970 ms to allow time for its arms to pan to the correct drum. This latency has been inherited from the existing firmware of DrumBot, and has been replicated within the framework's node. DrumBot possesses three arm modules, and thus can play three drums at any given time.

Despite this, there are still some playback limitations which MechSync addresses. DrumBot works based on a strict one-to-one mapping between MIDI note pitches and physical drum locations. This means that in practice, each drum can only be played by one arm, and will only respond to a single MIDI note. This can pose some issues when deciding which drums to map onto each arm. The standard MIDI specification provides mappings for a large number of drums which may or may not be present on DrumBot. Many drums specified are closely related, and so mapping these to the closest related drum on DrumBot can allow for much more flexibility in playback.

To achieve this, MechSync implements a mapping table for each arm of DrumBot, mapping each MIDI note to a virtual note. When an instruction is received, each arm's lookup table will be polled to see if it has an entry for that note. If an arm is found, the note is dispatched using the resulting mapping. If no appropriate arms are found, the note gets forwarded without mapping, and an error is logged to console. An additional benefit of this architecture is that it can allow for binding the same drum to multiple arms, with spacial-aware playback. This could be used in future DrumBot derivations to reduce the number of arms required for effective playback. In the case of the current hardware configuration, it has been used to compensate for the broken Arm 1 module, which has been absent for the duration of this project.

E. *PyNode*

For performance and MIDI processing requirements, MechSync is written in the Rust programming language. This allows for the use of existing VUW hardware, including lightweight systems such as Raspberry Pis. However, many developers of mechatronic instruments are unlikely to be experienced in Rust development, and may have difficulty expanding the software to work with their instrument. Additionally, understanding and manipulating the underlying synchronisation primitives used by May can be unintuitive.

For these reasons, MechSync provides the PyNode node, which allows for the integration of plug-ins written in Python. According to the TIOBE index, Python is currently the most popular programming language [29], and thus is an appropriate choice for integrating scripting support into compiled programs. Additionally, Python is much easier to write than Rust, with a plethora of supported library.

Unfortunately, Python is a rather poor choice for time-sensitive applications such as real-time audio processing. Due to the strong prevalence of the garbage collector, along with generally slow execution times, it is difficult to write python code which is temporally consistent. To mitigate this, PyNode wraps the executing Python module with a timing buffer, padding differences in execution durations. Execution time of each pass-through is measured, and a corresponding delay is applied to buffer all data to a given duration.

Another implication of calling Python code is that any blocking operations within the Python code will block the corresponding OS thread in the MechSync framework. This is undesirable, as blocking OS threads prevents other green-threads being swapped in, even if the blocking operation is just a call to sleep. To mitigate this, PyNode allows the Python code to return a sleep duration which can be applied after buffering. This means that instead of performing blocking sleep operations within Python, delays can be delegated to the Rust wrapper, which can then perform them asynchronously.

F. *DelayNode*

To allow for intra-node synchronisation within MIDI pipelines, MechSync provides the DelayNode node for buffering data by a chosen duration. This can operate as either a fixed delay, or as a desired total delay for the entire pipeline. When used as a total delay, the durations of all previous elements are calculated, and the DelayNode pads this up to the chosen latency. If the previous elements have a greater latency than specified by the DelayNode, an error will be reported to the user.

By utilising multiple DelayNode nodes over a graph with the same total duration, the output of each graph flow will become synchronous. Some nodes, such as DrumBot, provide an additional artificial delay value to DelayNode. This is to compensate for hardware latencies, as whilst DrumBot node does not buffer any MIDI instructions, the receiving hardware buffers all instructions for 1970 ms.

G. *I/O Nodes*

To interface with the outside MIDI environment of the host operating system, MechSync implements the producer and consumer nodes of InputNode and OutputNode. These nodes act as an abstraction around virtual MIDI ports, and are able to be dynamically constructed in order to add I/O to the graph.

MIDI Instructions are encoded using a variable-length binary format with close-packed data. MIDI utilises this to minimise memory overhead for instructions, which was a key restriction in early MIDI systems. Accessing values from a typical MIDI instruction requires bit-shifting and masks, which can be annoying to work with. MechSync resolves this by wrapping the MIDI data with the MidiData primitive type, which provides access to all fields of a MIDI instruction without shifts or masks.

InputNode utilises a call-back thread which listens for incoming MIDI instructions on a given OS MIDI port, handled by the Midir library. Upon receiving a MIDI instruction, the data is translated into the MidiData primitive type, and a new green-thread is spawned to process the data through the pipeline. When OutputNode receives data, it is converted back into the compact MIDI instruction format, and is forwarded to the Midir library to be dispatched to the virtual MIDI output.

IV. Implementation

A. *Programming Language*

Initial developments considered using the Kotlin programming language for this project. However, many factors resulted in Kotlin becoming unfavourable, and Rust being utilised instead. Firstly, Rust has much better support for developing applications working with MIDI data. Whilst Kotlin has access to the `javax.sound.midi` library, this library primarily focuses on MIDI file streaming. This would have limited MechSync to prestructured MIDI files, which would have made ad-hoc performance difficult. In contrast, Rust has access to real-time MIDI streaming libraries such as Midir, which are much more suited to the scope of this project.

Additionally, Rust provides better support for real-time programming in latency-dependent environments, due to the lack of garbage collection. With some implementations of the JVM (Java Virtual Machine), average garbage collection pauses of 16 ms could expected [30]. Given that our established tolerance for latency is 10 ms, this is unacceptable. Unlike Kotlin and other JVM languages, Rust has no virtual machine overhead. This is beneficial for achieving effective real-time performance on machines with limited processing power. By ensuring the project is able to run on existing hardware efficiently, we can further address the environmental and sustainability implications of this project.

B. *Midir*

The Midir [12] library acts as the key linkage between MechSync and the OS-level MIDI devices. Midir provides a highly capable API for creating and working with MIDI ports in Rust. Whilst Midir does support binding to existing MIDI ports and devices, it was decided that virtual MIDI ports were to be used instead. As MechSync targets Linux-based hosts, it is possible to take advantage of the modern Linux audio-stack via the Pipewire [31] system library. Pipewire was built as a unification of the JACK and alsa audio systems, and is designed for piping audio providers into audio consumers. Via the qpwgraph software, it becomes possible to use an intuitive UI to link together Pipewire components.

To take full advantage of this, virtual MIDI ports were chosen, as they allow Pipewire to handle the linkages between applications. This allows for MechSync ports to be bound and rebound on the fly, without requiring MechSync to rebind its own internal components. Additionally, this allows multiple MIDI sources and destinations to be connected to MechSync at once, including self-referencing loops.

Despite this flexibility, Midir's implementation does impose some limitations which introduces development challenges. On Linux, MIDI devices are represented as virtual files, which can be streamed to read or write MIDI data. Additionally, PipeWire provides APIs for creating and linking MIDI devices. Midir wraps these behaviours in a library API built around callbacks. Under the hood, each Midir port consists of a thread polling the MIDI queue for new instructions. As data arrives, it is segmented based on the MIDI status flags, and forwarded to the callback function. Unfortunately, this architecture has multiple implications.

Firstly, as Midir abstracts the MIDI queue from the library user, it is not possible to poll the queue without using an OS Thread. This means that callbacks from Midir cannot be built using asynchronous programming techniques such async and green-threads. This has some performance implications, although the cost is relatively minimal.

Additionally, the use of threads brings implications for soundness and memory safety. In earlier versions of Rust, threads were built using the RAII (Resource Acquisition Is Initialization) model of memory safety. Each thread guard has some lifetime in which the variable holding the guard lives. Threads therefore required that any references used within the thread must live at least as long as the thread itself, as otherwise, references could become stale and reference illegal memory. However, in what was later dubbed "Leakpocolypse", this approach was found be unsound. If the guard holding a thread context was leaked, and was never destructed, a thread could continue to run past the lifetime of its contents. To resolve this, modern Rust requires that all references within a thread live for `'static`, which represents the lifetime of the entire program. As Midir is built upon Rust threads, the callback function from a MIDI input must also be lifetime `'static`.

Unfortunately, this means that any references to child nodes in the graph must be lifetime `'static`, and therefore the entire node graph must be lifetime `'static`. This makes it difficult to dynamically create and remove MIDI ports, and makes dynamic configuration loading infeasible. To resolve this, InputNode wraps the Midir thread with an unsafe raw pointer to the `Weak` pointer of the next element within the graph. InputNode implements a custom Drop trait implementation, which ensures that the thread guard is always dropped before the pointer reference to the next element. As pointers are passed by value and are outside of the lifetime system, they are implicitly lifetime `'static`, and can therefore be used within threads. InputNode remains safe even if it is leaked, as both the pointer and the thread remain valid in memory.

C. May

MechSync uses the May [22] library to provide a friendly abstraction for green threading, simplifying mass processing of MIDI instructions. May implicitly handles yield points and dynamic call-sites, which allows programmers to focus on logical flow without considering the underlying threading architecture.

Originally, the asynchronous library Tokio [15] was utilised for development. Tokio provides a collection of backend asynchronous processors for the Rust async API. Tokio is highly performant, but suffers from high complexity due to the Rust's async implementation. Rust compiles asynchronous code into a fixed-size state machine. This is performed by traversing each function and building state changes at each function call. Unfortunately, this technique breaks when recursion is used. This is because it is impossible for the static analysis of the function calls to determine how deep the recursion will be required, resulting in an infinitely sized state machine. Additionally, dynamic callsites are difficult to utilise, as it is impossible for the state-machine to know what function will be called. Whilst Rust does provide solutions for these issues, fulfilling the lifetime promises of asynchronous futures means that the implementations are often obtuse. For these reasons, Tokio was dropped in favour of May.

May's `Thread::sleep` method is heavily utilised within MechSync for timing and scheduling MIDI instructions. As MechSync requires high temporal precision, it is important that any timing functions used within pipelines are accurate. Sleep operations within programs typically fallback onto kernel-level sleep calls, and therefore rely on the kernel for accuracy. On Windows-based operating systems, the kernel-level timer typically operates with 16 ms ticks. For MechSync, this would be unsuitable, as 16 ms is greater than the established minimum noticeable latency of 10 ms. In contrast, Linux kernels typically operate with 4 ms ticks, although this can vary based on distribution. Additionally, many Linux distributions such as Arch Linux feature support for high-resolution timers [32], which can provide nanosecond accuracy. As MechSync has been developed on Arch Linux, it takes advantage of high-resolution timers to ensure that playback is highly synchronous [33]. In systems where kernel sleep is inconsistent, spin-locking is often used. This involves looping over a conditional check, allowing for granularity at the level of CPU cycles, at the cost of high CPU overhead. Spin-locking was considered for MechSync, but found to be unnecessary, as the kernel timer was found to be sufficient.

May's `go` macro uses a similar syntax to the `std::thread::spawn` API, albeit with one major difference. May purposely does not handle leakpocalypse, instead choosing to wrap the `go` macro using an unsafe block. This means that technically, May uses an unsound API which can result in undefined behaviour. However, this requires purposely leaking the thread handler, which is highly unlikely to be done accidentally. May's `go` macro hides the thread handler's name from the programmer, so leaking the handler requires some effort. Additionally, May provides many warnings throughout the API against doing this. As a bonus, green-threads within May do not require `'static` lifetimes, and as such are much easier to work with compared to OS threads.

D. PyO3

To enable Python interop with MechSync, the PyO3 [34] library is utilised. This library provides an API for utilising Python bindings from within Rust programs. MechSync uses this functionality to allow for user-defined nodes within the MIDI graph. On startup, MechSync loads all bound Python modules within the YAML configuration file. This ensures that Python module loading and interpretation is preloaded, and thus does not affect the runtime performance of the program.

E. Serde

MechSync features dynamic graph construction via a loose-schema YAML configuration file. YAML parsing is handled via the serde_yaml library, which is built off the popular Serde library [35]. Serde supports a wide range of serialisation schemes such as JSON, TOML and YAML. Additionally, serialisation and deserialisation methods can often be automatically derived by utilising Serde's powerful code-generation macros.

MechSync uses Serde to handle constructing the graph from a given configuration. Loading is handled over two phases. Firstly, nodes are parsed into the backing graph using `Arc` smart pointers. A second pass is then performed to bind each node to its child within the call-graph, linking the nodes into MIDI processing chains. More specialised operations are delegated to node factories, which handle implementation-specific configuration such as DrumBot's drum assignments.

F. Hardware

The creation of MechSync involved working with a varied cast of mechatronic instruments, with diverse requirements for synchronisation and playback. These constraints heavily impacted the choices and methodologies chosen during this project. Ensuring the final artifact was highly flexible and adaptive would not only ensure effective playback with the current ensemble, but also ensures that MechSync is able to adapt to future instruments which VUW may create in the future.

The current implementation of MechSync focused on two of VUWs mechatronic instruments, MechBass and DrumBot. MechBass is a mechatronic bass guitar utilising four independent string modules. Each module features a panning pitch clamp used to select notes along the fret, and a pick-wheel used to play the given note. Modules are able to be independently controlled over MIDI by sending instructions to specific MIDI channels. DrumBot is a mechatronic drum robot built around a modified acoustic drum-kit. Drumbot consists of a kick-module, high-hat pedal module, and three arm modules which can pivot to play different drums. These arms must be programmed and calibrated before DrumBot can be used, which is typically done either via serial communication or the Drumbot GUI.

The development of MechSync also involved diagnosing and resolving multiple hardware failures which had accumulated over the lifespan of the robots. Initial tests with MechBass confirmed that three of the four modules were not functioning correctly. These modules had broken fret-clamps, which meant that the strings would always play the same note. This was due to broken solenoid wires, which had worn out after years of constant movement. These were replaced with new runs of wire, and as such, all four modules of MechBass are fully functional. Additionally, the existing foam pad dampeners on the fret clamps had completely disintegrated. These pads are important, as they absorb resonation in the top-ends of the strings, preventing dissonant overtones. The dampeners have been replaced with rubber pads, which consist of pieces of an old mousepad. This material was chosen as it is highly resistant to friction wear, whilst allowing the reuse of waste materials. By using existing VUW materials, and by finding unique methods of reusing waste, we can ensure that the maintenance of the mechatronic instruments is performed sustainably.

DrumBot has also been the focus of maintenance efforts, albeit focusing more on short-term perform-ability rather than long-term usage. Unfortunately, DrumBot's hardware has been constructed in a manner which has made efforts of restoration difficult. Much of DrumBot has been constructed using low-torque screws, which has resulted in many becoming chewed, requiring drilling to remove. Additionally, the design of DrumBot does not take repairability and maintenance into account, with key components such as the shoulder pivot requiring full deconstruction to access. This has hampered most efforts to repair DrumBot, as the time investment required to restore the hardware is beyond the scope of this project.

Currently, DrumBot consists of two working arms and the kick module. Arm 1 has been out of service for the duration of the project, and repairs have been deemed out of scope. Whilst the high-hat pedal module is present, the lack of schematics and apparent missing wires has made it difficult to reintegrate the module, and thus it remains disconnected. The overall wiring of DrumBot is messy and difficult to diagnose, and makes the robot impractical to transport. The kick module has also proven challenging to use, as the two solenoids are no longer matched in force, and often misalign and jam. This has been resolved temporarily by locking the load-balancer mechanism within the kick module using zip-ties. This prevents the module from pivoting when the solenoid force is lop-sided, which prevents it from jamming. In the near future, DrumBot will require full reconstruction, as any minor fixes performed on the hardware are unlikely to prevent the onset degradation of performance. Any future projects looking to replace DrumBot should investigate the current hardware limitations to diagnose and mitigate the failure modes which have plagued the current design.

V. Evaluation

A. Metrics

To evaluate the performance and capabilities of MechSync, a collection of metric-based and human-based tests have been performed. A key aspect of MechSync is its ability to synchronise multiple instruments with diverse playback latency characteristics. It is therefore important that latency between instruments is controlled and minimised, such that notes played concurrently over multiple instruments have no discernable latency.

The human ear does possess some tolerance for latency, which allows for some error when synchronising instruments. Utilising the Random Gap Detection Test, Owens et al. found that the minimum discernible latency was 9.23 – 12.15 ms [1]. From this research, we can define a desired maximum latency for MechSync of 10 ms.

An additional constraint for MechSync is that it should be capable of effective real-time playback of arbitrary MIDI instructions. A key characteristic of the existing mechatronic instruments is their ability to operate from live data on an ad-hoc basis. This provides a high degree of feedback and real-time interactivity, which makes working with the instruments flexible. MechSync should therefore perform minimal pre-buffering of instructions. Additionally, MechSync should also be capable of performing with a latency that matches the highest latency instrument within the ensemble. As DrumBot features an internal buffering of 1970 ms, MechSync should be capable of performing with this latency.

Finally, MechSync should be capable of producing a harmonious and pleasant listening experience. As this metric is based on human preference, this will be evaluated both by external users and by the author of MechSync. As a musician with 8 years of experience playing percussion instruments, they are appropriate as the pilot user for outlining the path for future studies involving MechSync. It should however be considered that utilising the author of MechSync as an evaluation metric does introduce some bias into the testing results. By utilising human heuristics, it is possible to evaluate MechSync against the more nuanced criterion of human experience. Additionally, due to the limitations of recording hardware available for testing, it is not possible to evaluate playback mechanisms such as MechBass's string dispatch with metric-based testing alone. Thus, utilising human testing allows for evaluation which can cover more aspects of the MechSync framework.

B. Test Methodologies

For evaluating MechSync against the latency requirements, a testing framework has been created to measure the temporal offset between the playback of MechBass and DrumBot. The framework operates by sending pairs of MIDI notes, and listening back for when the notes are played by the instruments. Each instrument is recorded using a Dayton Audio EMM-6 reference microphone, interfaced via a Focusrite 2i2 audio interface. Peaks within the recording signal are used to detect when a note is played. For DrumBot, the high-hat cymbals were chosen, as this produces a sharp and brief sound, which results in clearer peaks within the resulting recording waveform. For MechBass, the pick-wheel was recorded, as each played note results in a loud click as the picks make contact with the string. Timestamps are recorded for when the MIDI note is transmitted, and for when the designated microphone detects a playback peak. The difference in these two timestamps are saved as the latency of the instrument. From this, the data-points are then exported in a CSV format for analysis. This strategy was also used for collecting data to derive the latency characteristics of MechBass for curve derivation.

As the test operates using only two directional microphones, it is only possible to compare two modules per test. For the latency testing, it was decided to compare the top string module of MechBass with Arm 2 of DrumBot playing the high-hat, with the high-hat pedal absent. As each module of each instrument operates using the same firmware and hardware, proving the capability of synchronisation between singular modules implies that multiple modules should also play synchronously.

To ensure that MechSync is capable of maintaining latency over a range of MechBass intervals, the test uses a straddle pattern to cover a range of panning movements. This involves oscillating between the root and progressively increasing notes, beginning with no movement, and ending with a pan movement across the entire length of the string. These trials are performed five times to normalise any variance within the hardware configuration, including outliers.

C. Metric Results

Without MechSync, the latency between DrumBot and MechBass is highly apparent. As seen in Figure 2, when MechBass plays G2 $\rightarrow$ G2, there is no panning required. Thus, the difference in communication latency between the two instruments is approximately 1962 ms, close to the expected 1970 ms response time of DrumBot. Notably, the latency between the instruments reduces as the panning distance is increased. This is because MechBass takes longer to respond to notes that require further panning. In contrast, DrumBot contains its own synchronisation mechanisms, and as such, all communication with DrumBot is buffered by a constant value. However, even though the latency difference decreases, this is insignificant. With the maximum pan of G2 $\rightarrow$ G#3, the average latency of 1553 ms is still 155x larger than the maximum desired latency of 10 ms. Thus, without MechSync, the ensemble is highly unsynchronised.

In contrast, with MechSync enabled, the latency within the ensemble is reduced drastically. Considering the average latencies of each interval performed in Figure 3, all intervals except one fall within the desired $\pm$10 ms latency metric, and thus are synchronous. 19 out of 27 intervals tested have average latencies under 5 ms, with a distribution skewed towards smaller panning distances. This is expected, as smaller panning distances experience acceleration for a shorter duration, and thus feature less panning variation.

Only one interval's mean latency falls outside of the desired 10 ms tolerance, that being G2 $\rightarrow$ G#3, with an average latency difference of -11.6 ms. Fortunately, G2 $\rightarrow$ G#3 is

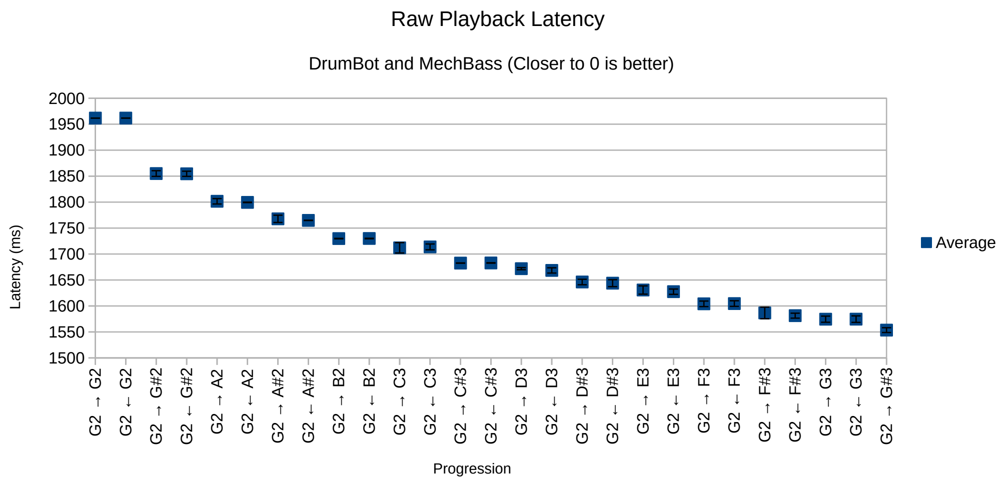

Fig. 2. Recorded latency without MechSync.

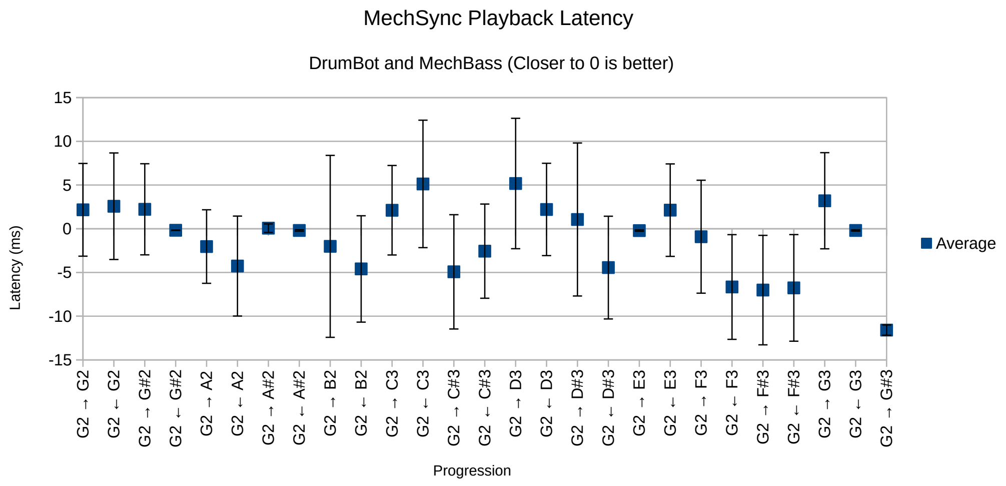

Fig. 3. Recorded latency with MechSync.

rarely performed within the normal operations of MechBass, as this interval requires panning over the entire length of the string module. Consequentially, an interval of this size would usually be dispatched over two strings to minimise panning distance.

MechSync uses deterministic functions to calculate synchronisation latencies. This means that any repeated intervals will result in MIDI buffering of the same duration. Consequentially, this means that any variance within the recorded data is a product of hardware variability, and not that of MechSync. This variability does pose some implications, as it limits how much MechSync can synchronise the ensemble. Inspecting the standard deviation error of G2 $\rightarrow$ B2 shows an error of $\pm$10.4 ms. This is larger than the $\pm$10 ms tolerance desired, and unfortunately means that there is no possible configuration for DrumBot and MechBass which can guarantee that all notes will be synchronous within 10 ms of each other. Fortunately, the majority of error bars fit within the upper bound of 12.15 ms as defined by Owens et al. [1]. All errors within the system are less than 15 ms, which are barely discernible. Within the context of a full mechatronic performance, it is unlikely that these would be noticed.

For the purposes of easier testing, a 2500 ms buffer was utilised to pad the differences in latency between MechBass and DrumBot. This can be seen within Figure 4, as both DrumBot and MechBass possess response times which hover around 2563 ms. The 63 ms discrepancy is due to audio buffering from testing code. As both MechBass and DrumBot are recorded through the same system, temporal comparisons are not affected, as this error is cancelled out. Additionally, MechBass is much less temporally consistent compared to DrumBot. This is likely due to DrumBot not being required to pan within this test. Future tests of MechSync may wish to test the panning latencies of both DrumBot and MechBass.

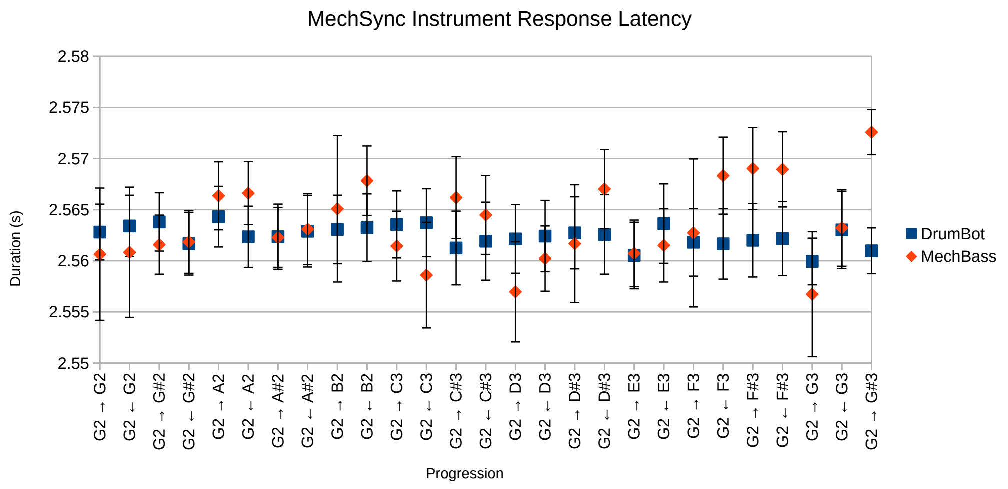

Fig. 4. Instrument latencies under MechSync with 2500 ms buffer.

D. Human Results

On Friday 23rd of August, VUW hosted its annual Open Day event. This event is performed to showcase the ongoing and previous developments occurring within the university, serving as a showcase of the university's work to prospective future students. As MechBass was usually showcased during the event, MechSync was graciously provided an opportunity to be evaluated within a public setting.

To showcase the capabilities of MechSync with MechBass, a MIDI rendition of Muse – Hysteria was used. This song features a fast bassline with highly technical instrumentation, fully exercising the scheduling and dispatch capabilities of the MechSync framework. Many listeners commented on how well MechBass handled playback, and were very interested in the underlying systems. No comments were made about any noticeable lag or stuttering during playback, which highlights the effectiveness of MechSync to minimise intra-instrument latencies. During the event, alternative songs were rotated in and out, including Gorillaz – Feel Good Inc. and Smash Mouth – All Star. For these songs, a virtual synthesiser Helm [36] was attached to MechSync, such that Helm and MechBass could play in tandem.

For additional interactivity, the MechSync input for MechBass was also attached to a MIDI keyboard. This allowed visitors to manually send notes to MechBass, and watch the dispatch mechanism in real-time. Despite a 406 ms latency between the MIDI keyboard and MechBass, many users still enjoyed the interactive element, and some managed to play full melodies. This shows that even with a high latency, the novelty of interacting with MechBass in a live setting is still enjoyable.

Based on the author's personal evaluation of MechSync, they find it to be a highly capable framework. They are unable to perceive any latency drift or temporal issues during standard playback. The dynamic MIDI processing makes it highly interactive, and allows it to play a wide range of music. Even in instances where the MechBass node's dispatching is unable to keep up with a given track, such as Opeth – By the Pain I See in Others, it is able to recover quickly, and the listening experience is still very enjoyable. Future studies may wish to evaluate this heuristic using a wider range of independent qualified evaluators.

VI. Conclusion and Future Work

A. Limitations

Despite performing highly effectively as a synchronisation framework, MechSync still features some key limitations which have reduced the scope of what could have been developed. MechSync operates as a purely MIDI-oriented framework, with all external communications occurring over virtual MIDI devices. Unfortunately, MIDI is limited in its scope of expressiveness and playback, as it is not designed to handle diverse playback techniques. Additionally, any non-MIDI communication to instruments must occur externally to MechSync. This means that external software must be used for instruments that require calibration, such as DrumBot. To resolve this, future iterations of MechSync could allow for multiple modes of communication, such I2C. Communication with MechSync could then be handled via a VST API, such as the Steinburg vst3sdk [37].

The development and evaluation of MechSync has also been impacted by the hardware conditions of the available mechatronic instruments. DrumBot has proven particularly difficult to work with, due to unreliable firmware, and numerous hardware failures. At the time of MechSync's evaluation, DrumBot is unable to use any of its three arms to full capacity, with both available arms seizing up when panning. This has made it difficult to showcase and evaluate MechSync, as the unreliable performance of DrumBot has made impossible to showcase nuanced playback. During development, considerations were made to integrate the guitar robot Azure Talos into the mechatronic ensemble. Unfortunately, Azure Talos has been unavailable during development, and so it was never integrated into the project. Fortunately, Azure Talos behaves similarly to MechBass, and so any future efforts to integrate Azure Talos should be able to utilise the existing MechBass code and architecture.

B. Future Work

The developments of MechSync have paved the way for considerable future developments within the mechatronic instrumentation space. By providing a reusable and flexible framework for implementing synchronisation within mechatronic ensembles, future instruments can utilise MechSync capabilities, and thus can be rapidly integrated with the existing mechatronic ensemble. This is further promoted via the PyNode API, which allows future instrument creators to develop for MechSync using Python, without concerns of the implementation details of synchronisation.

As MechSync handles inter-instrument and intra-instrument synchronisation, this allows future developments with the mechatronic ensemble to focus on other aspects of playback such as dynamic composition. Novel compositional techniques using generative AI could be developed, which would allow for live improvisational performance. As the AI would not have to consider the temporal limitations of the attached hardware, developments would be able to focus more on discovering interesting compositional strategies.

By processing MIDI data in real-time, MechSync also allows for live MIDI playback of arbitrary compositions. This enables MechSync to be used in live contexts such as events and concerts. Additionally, the framework can even support human-controlled performance ad-hoc, albeit with some latency introduced by the synchronisation process. This further expands the opportunities for current and future instruments to be toured.

C. Project Achievements

Overall, this project has successfully created an artifact capable of synchronising a mechatronic ensemble with a diverse range of musical progressions. The MVP requirements for this project outline that it should be capable of playing one string of MechBass synchronously with one drum of DrumBot. The resulting MechSync software has far surpassed these capabilities as outlined, and is capable of utilising the full feature set of both MechBass and DrumBot, whilst maintaining synchronous playback. Additionally, MechSync is not only built to handle current instruments, but has been developed to be flexible and expandable to any future developments of VUW instruments. The latencies of instruments under MechSync have proven to be below the tolerance detectable to human listeners, and thus MechSync successfully meets the latency requirements outlined. Given the work that has been completed, it is likely that MechSync will continue to see use far beyond the development of this project, as VUW continues to push the envelope with its mechatronic instrument developments.

REFERENCES

[1] D. Owens, P. E. Campbell, A. Liddell, C. DePlacido, and M. Wolters. (2007) Random gap detection test: A useful measure of auditory ageing? Accessed: 2024-05-24. [Online]. Available: https://www.cs.stir.ac.uk/~kjt/research/match/resources/documents/efas07-owens.pdf

[2] M. Sithu, Y. Ishibashi, and N. Fukushima, "Dynamic local lag control for sound synchronization in joint musical performance," in *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, Dec 2013.

[3] S. Park and Y. Choi, "Real-time multimedia synchronization based on delay offset and playout rate adjustment," *Real-Time Imaging*, vol. 2, no. 3, pp. 163–170, Jun 1996.

[4] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, Apr 2007.

[5] PortMedia, "Portmedia," accessed: 2024-05-24. [Online]. Available: https://portmedia.sourceforge.net/

[6] The Audacity Team, "Audacity: Free audio editor, recorder, music making and more!" accessed: 2024-05-24. [Online]. Available: https://www.audacityteam.org/

[7] Oracle, "Java development kit version 22 api specification," accessed: 2024-05-24. [Online]. Available: https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/sound/midi/package-summary.html

[8] L. Gorven, "General midi standards: Table 1 - instrument patch map," accessed: 2024-05-24. [Online]. Available: https://www.cs.cmu.edu/~music/cmsip/readings/GMSpecs_Patches.htm

[9] Alsa Opensrc Org, "Playing midi files," accessed: 2024-05-24. [Online]. Available: https://alsa.opensrc.org/PlayingMIDIFiles

[10] T. Erichsen, "virtualmidi sdk," accessed: 2024-05-24. [Online]. Available: https://www.tobias-erichsen.de/software/virtualmidi/virtualmidi-sdk.html

[11] G. P. Scavone, "The rtmidi tutorial," accessed: 2024-05-24. [Online]. Available: http://www.music.mcgill.ca/~gary/rtmidi/

[12] P. Reisert, "Boddlnagg/midir: Cross-platform realtime midi processing in rust." accessed: 2024-05-24. [Online]. Available: https://github.com/Boddlnagg/midir

[13] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan 2010.

[14] P. Oppermann, "Async/await," accessed: 2024-05-24. [Online]. Available: https://os.phil-opp.com/async-await/

[15] Tokio, "Build reliable network applications without compromising speed," accessed: 2024-05-24. [Online]. Available: https://tokio.rs/

[16] M. Kline, "Async rust is a bad language," accessed: 2024-05-24. [Online]. Available: https://bitbashing.io/async-rust.html

[17] B. Nystrom, "What color is your function?" accessed: 2024-05-24. [Online]. Available: https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

[18] Go, "Goroutines," accessed: 2024-05-24. [Online]. Available: https://go.dev/tour/concurrency/1

[19] University of Alberta, "Understanding threads," accessed: 2024-05-24. [Online]. Available: https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixprggd/genprogc/understanding_threads.htm

[20] JetBrains, "Coroutines," accessed: 2024-05-24. [Online]. Available: https://kotlinlang.org/docs/coroutines-overview.html

[21] H. Yabuuchi, "rfcs/text/0230-remove-runtime.md at master · rust-lang/rfcs," accessed: 2024-05-24. [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0230-remove-runtime.md

[22] X. Huang, "Xudong-huang/may: Rust stackful coroutine library," accessed: 2024-05-24. [Online]. Available: https://github.com/Xudong-Huang/may

[23] TechEmpower, "Techempower web framework benchmarks," accessed: 2024-05-24. [Online]. Available: https://www.techempower.com/benchmarks/#hw=ph&test=composite&section=data-r22

[24] J. F. et al. (2021) Pipeline trees - an auxiliary tool in the creation of time series pipelines. Accessed: 2024-05-24. [Online]. Available: https://ix.cs.uoregon.edu/~jmclaug2/manuscripts/pub/flores2021pipeline.pdf

[25] E. Lee and D. Messerschmitt, "Pipeline interleaved programmable dsp's: Synchronous data flow programming," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 9, pp. 1334–1345, Sep 1987.

[26] D. C. Schmidt, "How java parallel streams work "under the hood"," accessed: 2024-05-24. [Online]. Available: http://www.dre.vanderbilt.edu/~schmidt/cs253/2022-PDFs/6.2.2-overview-of-Java-parallelstreams-phases.pdf

[27] S. , "Generators," mar 26 2023, [Accessed: 2024-10-12]. [Online]. Available: https://without.boats/blog/generators/

[28] Rust, "Generator in std::ops," [Accessed: 2024-10-12]. [Online]. Available: https://dev-doc.rust-lang.org/beta/std/ops/trait.Generator.html

[29] P. Jansen, "Tiobe Index," dec 27 2021, [Accessed: 2024-10-12]. [Online]. Available: https://www.tiobe.com/tiobe-index/
[30] Oracle, "Garbage Collection Performance," [Accessed: 2024-10-12]. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/performissues003.html
[31] Pipewire, "Pipewire," [Accessed: 2024-10-13]. [Online]. Available: https://pipewire.org/
[32] Embedded Linux Wiki, "High Resolution Timers," [Accessed: 2024-10-12]. [Online]. Available: https://www.elinux.org/High_Resolution_Timers
[33] Arch Linux, "x86_64_defconfig," [Accessed: 2024-10-12]. [Online]. Available: https://www.github.com/archlinux/linux/blob/98f7e32f20d28ec452afb208f9cffc08448a2652/arch/x86/configs/x86_64_defconfig
[34] PyO3, "Github - PyO3/pyo3: Rust bindings for the Python interpreter," [Accessed: 2024-10-13]. [Online]. Available: https://github.com/PyO3/pyo3
[35] Serde, "Overview · Serde," [Accessed: 2024-10-13]. [Online]. Available: https://serde.rs/
[36] M. Tytel, "Helm - Free Synth by Matt Tytel," [Accessed: 2024-10-13]. [Online]. Available: https://tytel.org/helm/
[37] Steinburg, "Github - steinbergmedia/vst3sdk: Vst 3 Plug-In SDK," [Accessed: 2024-10-12]. [Online]. Available: https://www.github.com/steinbergmedia/vst3sdk