1 Abstract

Explain briefly the paper and what it does.

2 Introduction

Scientific Workflow Management Systems (SWMSs) are an essential tool for automating, managing, and executing complex scientific processes involving large volumes of data and computational tasks[1]. Traditional SWMSs employ a linear sequential approach, in which tasks are performed in a pre-defined order, as defined by the workflow. While this linear method is suitable for certain applications, it might not always be the best choice: processing sequentially can prove inefficient in cases where the next step of the process should adapt to the previous one. For these use-cases a dynamic scheduler is required, of which *Managing Event Oriented Workflows*[2] (MEOW) is one.

Expand on DAGs' inability to adapt. Plagiarize David's thesis.

MEOW employs an event-based scheduler, in which jobs are performed non-linearly (**Better word here**), triggered based on events[2]. By dynamically adapting the execution order based on the outcomes of previous tasks or external factors, MEOW provides a more efficient and flexible solution for processing large volumes of experimental data[3].

- Expand on what "efficient" is
- What work am I doing on MEOW?
- How did it go?
- Introduce the concept of network events.
- **Write this last**

2.1 Problem

In its current implementation, MEOW is able to trigger jobs based on changes to monitored local files. This covers a the range of scenarios where the data processing workflow involves the creation, modification, or removal of files. By monitoring file events, MEOW's event-based scheduler can dynamically execute tasks as soon as the required conditions are met, ensuring efficient and timely processing of the data. Since the file monitor is triggered by changes to local files, MEOW is limited to local workflows.

[1] citation?
[2] citation?
[3] citation?

While file events work well as a trigger on their own, there are several scenarios where a different trigger would be preferred or even required, especially when dealing with distributed systems or remote operations. To address these shortcomings and further enhance MEOW's capabilities, the integration of network event triggers would provide significant benefits in several key use-cases.

Firstly, network event triggers would allow for manual triggering of jobs remotely, without the need for direct access to the monitored files. This is particularly useful in human-in-the-loop scenarios, where human intervention or decision-making is required before proceeding with the subsequent steps in a workflow. While it is possible to manually trigger job using file events by making changes to the monitored directories, this might lead to an already running job accessing the files at the same time, which could cause problems with data integrity.

Secondly, incorporating network event triggers would facilitate seamless communication between parallel runners, ensuring that tasks can efficiently exchange information and updates on their progress, allowing for a better perspective on the whole workflow, greatly improving visibility and control.

Finally, extending MEOW's event-based scheduler to support network event triggers would enable the simple and efficient exchange of data between workflows running on different machines. This feature is particularly valuable in distributed computing environments, where data processing tasks are often split across multiple systems to maximize resource utilization and minimize latency.

Integrating network event triggers into MEOW would provide an advantage specifically in the context of heterogeneous workflows, which incorporate a mix of different tasks running on diverse computing environments. By their nature, these workflows can involve tasks running on different systems, potentially even in different physical locations, which need to exchange data or coordinate their progress. In the figure below, an example heterogeneous workflow is presented.

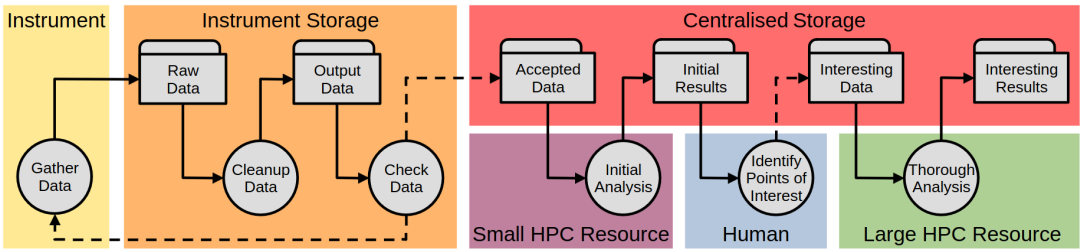

Figure 1: An example of a heterogeneous workflow

The example workflow requires several checkpoints in which data should be transferred between the instrument, the instrument storage, centralized storage, High Performance Computing (HPC) resources, and a human interaction point. Network events can, for the reasons outlined earlier in the section, be used to prevent the workflow from halting when these points are reached.

2.2 Background

2.2.1 The structure of MEOW

The MEOW event-based scheduler consists of four main components: *monitors*, *handlers*, *conductors*, and *the runner*.

Monitors listen for triggering events. They are initialized with a number of *rules*, which each include a *pattern* and *recipe*. *Patterns* describe the triggering event. For file events, the patterns describe a path that should trigger the event when changed. *Recipes* describe the specific action that should be taken when the rule is triggered. When a pattern's triggering event occurs, the monitor sends an event, which contains the rule and the specifics of the event, to the event queue.

Handlers manage the event queue. They unpack and analyze events in the event queue. If they are valid, a job is created from the recipe, which is then sent to the job queue.

Conductors manage the jobs queue. They execute the jobs that have been created by the handlers.

Finally, the runner is the main program that orchestrates all these components. Each instance of the runner incorporates at least one instance of a monitor, handler, and conductor, and it holds the event and job queues.

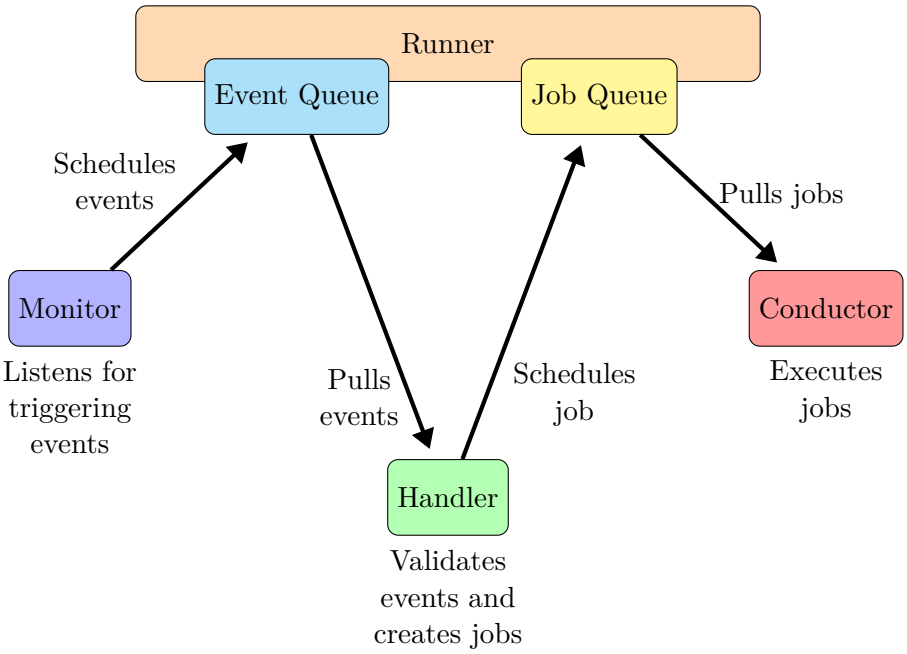

Figure 2: How the elements of MEOW interact

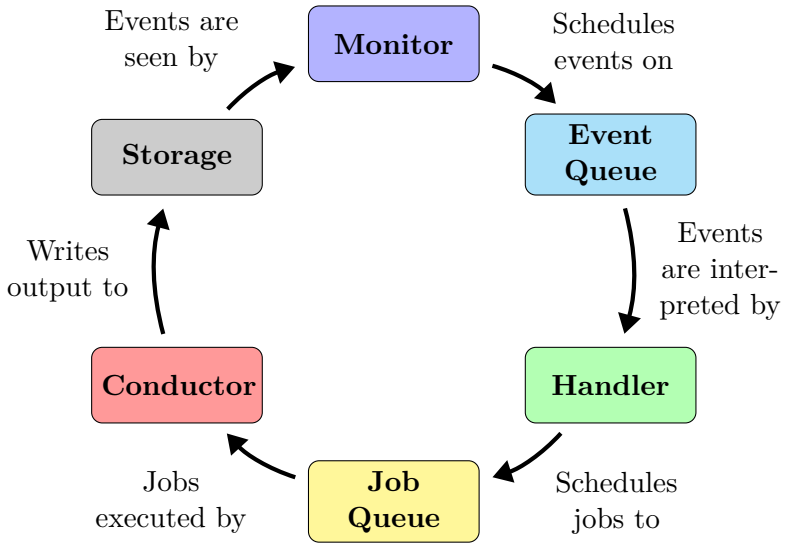

Figure 3: The cycle of MEOW's file events

2.2.2 The `meow_base` codebase

`meow_base`[3] is an implementation of MEOW written in python. It is written to be modular, using base classes for each element in order to ease the implementation of additional handlers, monitors, etc.

The relevant parts of the implementation are:

- **Events** are python dictionaries, containing the following items:
  - `EVENT_PATH`: The path of the triggering file.
  - `EVENT_TYPE`: The type of event, e.g. `"watchdog"`.
  - `EVENT_RULE`: The rule that triggered the event, which contains the recipe that the handler will turn into a job.
  - `EVENT_TIME`: The timestamp of the triggering event.
  - Any extra data supplied by the monitor. File events are by default initialized with the base directory of the event and a hash of the event's triggering path.
- **The file event monitor** inherits from the `BaseMonitor` class. It uses the `Watchdog` module to monitor given directories for changes. The Watchdog monitor is initialized with an instance of the `WatchdogEventHandler` class as its event handler. When the Watchdog monitor is triggered by a file event, the `handle_event` method is called on the event handler, which in turn creates an `event` based on the specifics of the triggering event. The event is then sent to the runner to be put in the even queue.
- **The runner** is implemented as the class `MeowRunner`. When initialized with at least one instance of a monitor, handler, and conductor, it validates them. When started, all the monitors, handlers, and conductors it was initialized with are started. It also creates `pipes` for the communication between each element and the runner.
- **Recipes** inherit from the `BaseRecipe` class. They mainly exist to contain data about a given recipe, but also contain validation checks.

- **Handlers** inherit from the `BaseHandler` class. Handler classes are for a specific type of job, like the execution of bash scripts. When started, it enters an infinite loop, where it asks the runner for a valid event in the event queue, and then creates a job for the recipe, and sends it to the runner to put in the job queue.
- **Conductors** inherit from the `BaseConductor` class. Conductor classes are for a specific type of job, like the execution of bash scripts. When started, it enters an infinite loop, where it asks the runner for a valid job in the job queue, and then attempts to execute it.

2.2.3 The `socket` library

The `socket` library[1], included in the Python Standard Library, serves as an interface for the Berkeley sockets API. The Berkeley sockets API, originally developed for the Unix operating system, has become the standard for network communication across multiple platforms. It allows programs to create 'sockets', which are endpoints in a network communication path, for the purpose of sending and receiving data.

Many other libraries and modules focusing on transferring data exist for Python, some of which may be better in certain MEOW use-cases. The `ssl` library, in specific, allows for ssl-encrypted communication, which may be a requirement in workflows with sensitive data. However, implementing network triggers using the `socket` library will provide MEOW with a fundamental implementation of network events, which can later be expanded or improved with other features.

In my project, all sockets use the Transmission Control Protocol (TCP), which ensures safe data transfer by enforcing a stable connection between the sender and receiver.

I make use of the following socket methods, which have the same names and functions in the `socket` library and the Berkeley sockets API:

- `bind()`: Associates the socket with a given local IP address and port. It also reserves the port locally.
- `listen()`: Puts the socket in a listening state, where it waits for a sender to request a TCP connection to the socket.
- `accept()`: Accepts the incoming TCP connection request, creating a connection.
- `recv()`: Receives data from the given socket.
- `close()`: Closes a connection to a given socket.

During testing of the monitor, the following methods are used to send data to the running monitor:

- `connect()`: Sends a TCP connection request to a listening socket.
- `sendall()`: Sends data to a socket.

3 Method

To address the identified limitations of MEOW and to expand its capabilities, I will be incorporating network event triggers into the existing event-based scheduler, to

supplement the current file-based event triggers. My method focuses on leveraging Python's socket library to enable the processing of network events. The following subsections detail the specific methodologies employed in expanding the codebase, the design of the network event trigger mechanism, and the integration of this mechanism into the existing MEOW system.

3.1 Design of the network event pattern

In the implementation of a pattern for network events, a key consideration was to integrate it seamlessly with the existing MEOW codebase. This required designing the pattern to behave similarly to the file event pattern when interacting with other elements of the scheduler. A central principle in this design was maintaining the loose coupling between patterns and recipes, minimizing direct dependencies between separate components. While this might not be possible for every theoretical recipe and pattern, designing for it could greatly improve future compatibility.

The `NetworkEventPattern` class is initialized with a triggering port, analogous to the triggering path used in file event patterns. This approach inherently limits the number of unique patterns to the number of ports that can be opened on the machine. However, given the large number of potential ports, this constraint is unlikely to present a practical issue. An alternative approach could have involved triggering patterns using a part of the sent message, essentially acting as a "header". However, this would complicate the process since the monitor is otherwise designed to receive raw data. To keep the implementation as straightforward as possible and to allow for future enhancements, I opted for simplicity and broad utility over complexity in this initial design.

When the `NetworkMonitor` instance is started, it starts a number of `Listener` instances, equal to the number of ports specified in its patterns. Patterns not associated with a rule are not considered, since they will not result in an event. Only one listener is started per port, so patterns with the same port use the same listener. The listeners each open a socket connected to their respective ports. This is consistent with the behavior of the file event monitor, which monitors the triggering paths of the patterns it was initialized with.

3.2 Integrating network events into the existing codebase

The data received by the network monitor is written to a temporary file; this design choice serves three purposes:

Firstly, this method is a practical solution for managing memory usage during data transfer, particularly for large data sets. By writing received data directly to a file, we bypass the need to store the entire file in memory at once, effectively addressing potential memory limitations.

Secondly, the method allows the monitor to receive multiple files simultaneously, since receiving the file will be done by separate threads. This means that a single large file will not "block up" the network port for too long.

Lastly, this approach allows the leveraging of existing infrastructure built for file

events. The newly written temporary file is passed as the "triggering path" of the event, mirroring the behavior of file events. This approach allows network events to utilize the recipes initially designed for file events without modification, preserving the principle of loose coupling. This integration maintains the overall flexibility and efficiency of MEOW while extending its capabilities to handle network events.

The method will be slower, since writing to storage takes longer than keeping the data in memory, but I have decided that the positives outweigh the negatives.

3.3 Testing

The unit tests for the network event monitor were inspired by the already existing tests for the file event monitor. Since the aim of the monitor was to emulate the behavior of the file event monitor as closely as possible, using the already existing tests with minimal changes proved an effective way of staying close to that goal.

4 Results

The testing suite designed for the monitor comprised of 26 distinct tests, all of which successfully passed. These tests were designed to assess the robustness, reliability, and functionality of the monitor. They evaluated the monitor's ability to successfully manage network event patterns, detect network events, and communicate with the runner to send events to the event queue.

4.1 Discussion

With the hindsight of the results, what could I have done better?

4.2 Future Work

4.2.1 Use-cases for Network Events

Since the purpose of the project was adding a feature to a workflow manager, it's important to consider its integration within real-life workflows and consider future workflow designs that will capitalize on Network Events.

One specific example of an application where network event triggers could prove useful is the workflow for The Brain Imaging Data Structure (BIDS). The BIDS workflow requires data to be sent between multiple machines and validated by a user. Network event triggers could streamline this process by automatically initiating data transfer tasks when specific conditions are met, thereby reducing the need for manual management. Additionally, network triggers could facilitate user validation by allowing users to manually prompt the continuation of the workflow through specific network requests, simplifying the user's role in the validation process.

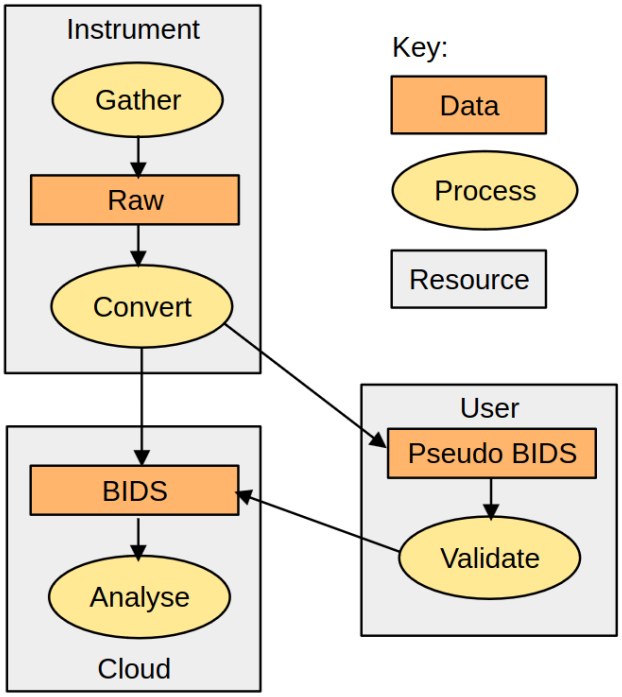

Figure 4: The structure of the BIDS workflow. Data is transferred to user, and to the cloud.

4.2.2 Additional Monitors

The successful development and implementation of the network event monitor for MEOW serves as a precedent for the creation of additional monitors in the future. This framework could be utilized as a blueprint for developing new monitors tailored to meet specific demands, protocols, or security requirements.

For instance, security might play a crucial role in the processing and transfer of sensitive data across various workflows. The network event monitor developed in this project, which uses the Python socket library, might not satisfy the security requirements of all workflows, especially those handling sensitive data. In such cases, developing a monitor that leverages the `ssl` library could provide a solution, enabling encrypted communication and thus improving the security of data transfer. The architecture of the network event monitor can guide the development of an `ssl` monitor, taking advantage of the similarities between the `socket` and `ssl` libraries.

Similarly, we could envision monitors developed specifically for certain protocols. For example, a monitor designed to handle HTTP requests could be beneficial for workflows interacting with web services. As HTTP is a common protocol, this type of monitor would open up a vast array of potential interactions with external services, making MEOW even more versatile.

5 Conclusion

With the monitor performing effectively as tested, it can be anticipated that it will handle network event triggers correctly in live environments. This is a critical enhancement for MEOW, opening up possibilities for more complex, distributed, and heterogeneous workflows, as envisioned in the design objectives.

References

[1] Python documentation. *socket - Low-level networking interface*. https://docs.python.org/3/library/socket.html.

[2] David Marchant. "MEOW - Enabling Dynamic Scheduling of Scientific Analysis". PhD thesis. University of Copenhagen, May 2021.

[3] David Marchant. *meow_base*. https://github.com/PatchOfScotland/meow_base. 2023.