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

MEOW employs an event-based scheduler, in which jobs are performed non-linearly, 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].

- 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.

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

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

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 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 synchronize their progress.

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. Currently, MEOW's reliance on local file events as triggers can be a limiting factor in these scenarios. Network event triggers offer a powerful solution to this challenge. They can not only handle tasks running across different machines, but also dynamically adapt to the changing requirements of a heterogeneous workflow, such as triggering new tasks based on the results of remote computations. Thus, the addition of network event triggers is a significant step in enhancing MEOW's already robust handling of heterogeneous workflows, bolstering its utility in today's diverse and distributed computing landscape.

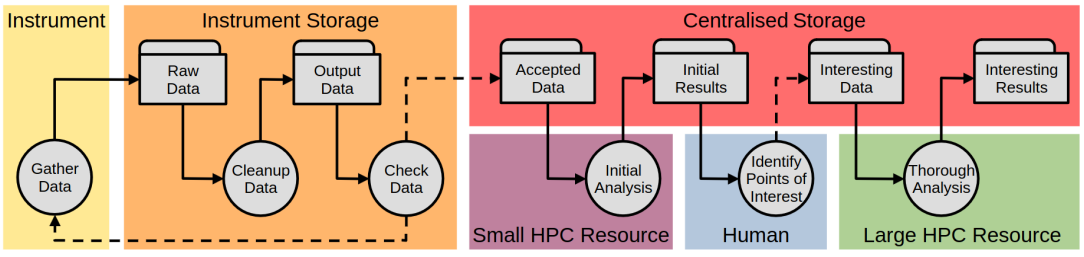

Figure 1: An example of a heterogeneous workflow

2.2 Background

2.2.1 The structure of MEOW

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

Monitors listen for triggering events. They are initialized with a number of *patterns*, which describe the triggering event. When a pattern's triggering event occurs, the monitor signals to the conductor that the pattern has been triggered, and schedules a job that has been associated with the pattern.

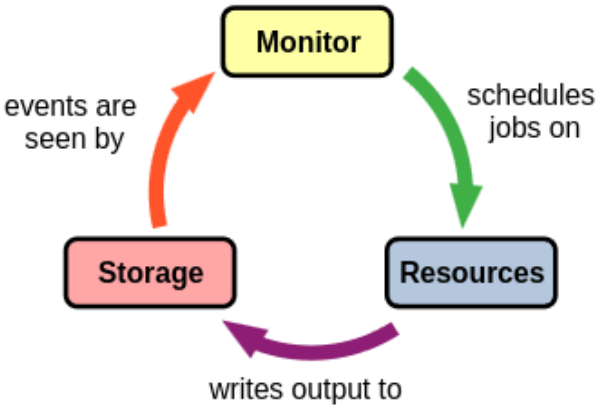

Figure 2: The monitor's role in MEOW's event-based system.

I haven't used "Resources" to describe the job queue. Should I do that or should I rephrase the diagram to be more in line with the rest of the project?

Handlers perform actions and jobs on behalf of the scheduler. They are initialized with a number of *recipes*, which describe the action to be taken. The handler starts a job when signal to do so by the conductor.

The conductor handles the jobs queue. It is initialized with a number of rules, which a pattern paired with a recipe. When a monitor sends it a triggered pattern, the rules are checked for that pattern. If one or more rules contain that pattern, the corresponding recipes are triggered in their handler.

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.

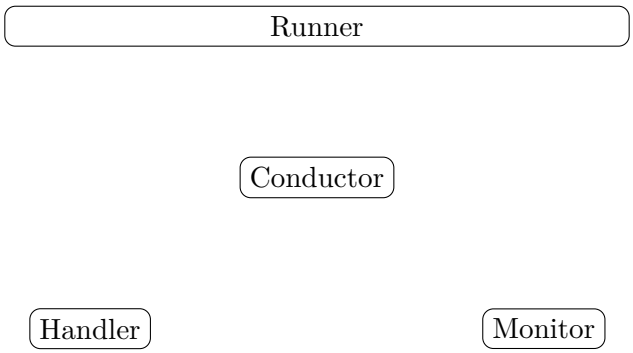

Figure 3: **WIP.** How the elements of MEOW interact.

2.2.2 The `meow_base` codebase

Specific (but not too granular) implementation details of `meow_base`.

The current implementation of MEOW, `meow_base`[3], …

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 basic 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:

Too granular?

- `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.

- `connect()`: Sends a TCP connection request to a listening socket. This is only used in testing the monitor.
- `sendall()`: Sends data a socket. This is only used in testing the monitor.
- `close()`: Closes a connection to a given 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

A main concern with implementing a pattern for network events is to seamlessly integrate it with the existing codebase. Because of this, the design of the pattern has a heavy focus on behaving similarly to the file event pattern when interacting with the other elements of the scheduler. Ideally, this should preserve loose coupling of the patterns and recipes, so any pattern can be put in a rule with any recipe. While this might not be possible for every theoretical recipe and pattern, designing for it could greatly improve future compatibility.

Network event patterns are initialized with a triggering port, similar to the triggering path of the file event patterns. While this limits the amount of possible unique patterns to the amount of ports that can be opened on the machine, that amount is large enough that it will likely not be an issue. It would have been possible to have the patterns be triggered by part of the sent message, acting as a "header". However, this would complicate the process, since the monitor will otherwise be expecting to receive raw data. This was chosen in order for the implementation to be as simple as possible, so that any feature or improvement can be added later as its own pattern type.

The network monitor, when started, opens sockets that start listening on the ports specified in the patterns it was initialized with.

3.2 Integrating it into the existing codebase

Data received by the network monitor is written to a temporary file, which serves two purposes. Firstly, writing the received data to a file while receiving it saves on memory, since the entire file doesn't have to be saved in memory at once. This is especially useful for large data transfers. Secondly, writing the received data to a file allows network events to reuse most of the infrastructure written for file events, passing the newly written temporary file as the "triggering path" of the event. This means that recipes taking the triggering path as their input can still be used with network events, preserving loose coupling.

3.3 Testing

4 Results

Does it work? How well?

4.1 Discussion

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

4.2 Future Work

What should someone do if they want to fix my mistakes, or expand on them further.

- Implementation of the other options mentioned when discussing the socket library.
- Triggering on a header item in addition to port

Give context to following paragraph.

One specific example of a use-case 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.

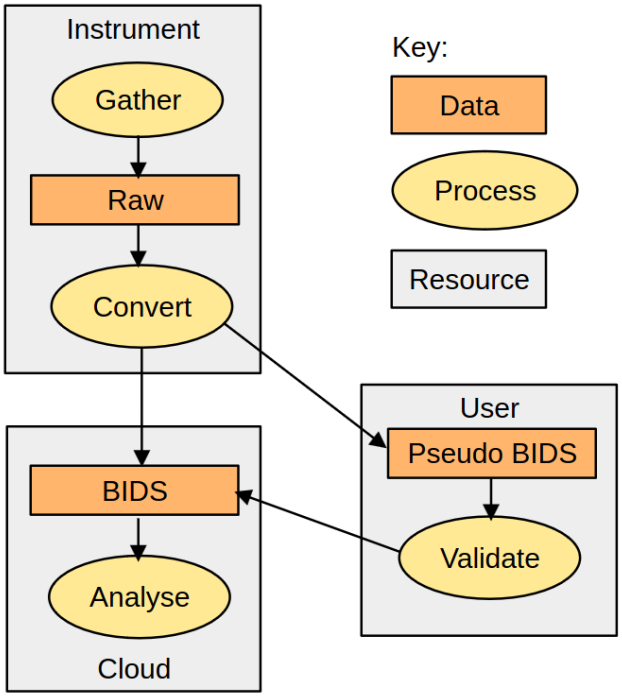

Figure 4: **Temp.** The structure of the BIDS workflow. Data is transferred to user, and to the cloud.

5 Conclusion

Did I succeed in what I wanted to do?

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.