Adding Network Event Triggers to an Event-based Workflow Scheduler

Nikolaj Ingemann Gade (qhp695)

June 2023

1 Abstract

This paper introduces a network event monitor to the Managing Event Oriented Workflows (MEOW) system, enabling it to respond to data transmitted over a network connection. The Python-based implementation uses the socket library, incorporates a new pattern type for network events, and reuses existing infrastructure for file events. Performance tests reveal robust handling of events with multiple listeners, demonstrating the viability of this enhancement. The design fosters future extensions, marking an essential step in advancing the capabilities of scientific workflow management systems to meet the dynamic demands of data-intensive fields

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. Jobs in a SWMS workflows are typically defined as the nodes in a Directed Acyclic Graph (DAG), where the edges define the dependencies of each job.

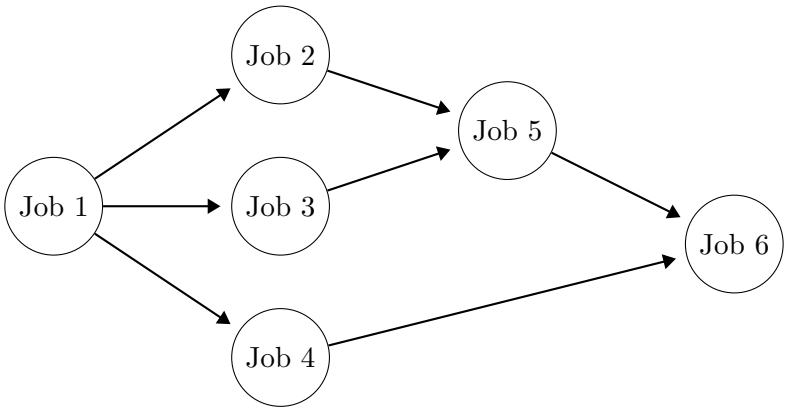

Figure 1: A workflow defined as a DAG. Job 2, 3, and 4 are dependent on the completion of Job 1, etc.

While this method is suitable for many applications, it may not always be the best solution. Processing the jobs in a set order can lead to inefficiencies in cases where the processing of the jobs needs to adapt based on the results of earlier jobs, human interaction, or changing circumstances. In these contexts, the DAG method might fall short due to its inherently static nature.

In such scenarios, using a *dynamic scheduler* can offer a more effective approach. Unlike traditional DAG-based systems, dynamic schedulers are designed to adapt dynamically to changing conditions, providing a more adaptive method for managing complex workflows. One such dynamic scheduler is the *Managing Event Oriented Workflows*[4] (MEOW).

MEOW employs an event-based scheduler, in which jobs are executed independently, based on certain *triggers*. Triggers can in theory be anything, but are currently limited to file events on local storage. By dynamically adapting the execution order based on the outcomes of previous tasks or external factors, MEOW provides a more flexible solution for processing large volumes of experimental data, with minimal human validation and interaction[3].

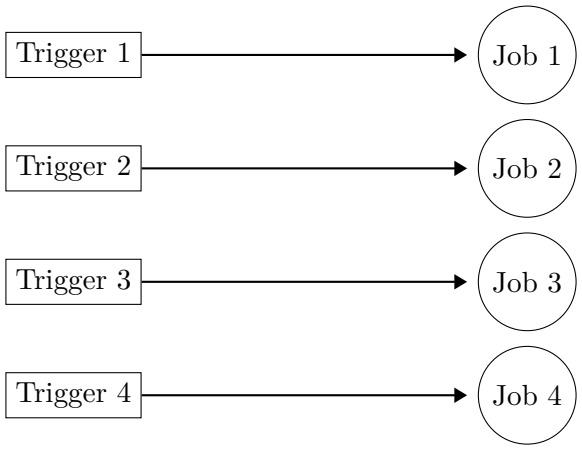

Figure 2: A workflow using an event-based system. Job 1 is dependent on Trigger 1, etc.

In this project, I introduce triggers for network events into MEOW. This enables a running scheduler to react to and act on data transferred over a network connection. By incorporating this feature, the capability of MEOW is significantly extended, facilitating the management of not just local file-based workflows, but also complex, distributed workflows involving communication between multiple systems over a network.

In this report, I will walk through the design and implementation process of this feature, detailing the challenges encountered and how they were overcome.

2.1 Problem

In its current implementation, MEOW is able to trigger jobs based on changes to monitored local files. This covers a 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 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 enable the initiation of jobs remotely through the transmission of a triggering message to the monitor, thereby eliminating the necessity 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 workflows, ensuring that tasks can efficiently exchange information and updates on their progress, allowing for a better perspective on the combined 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.

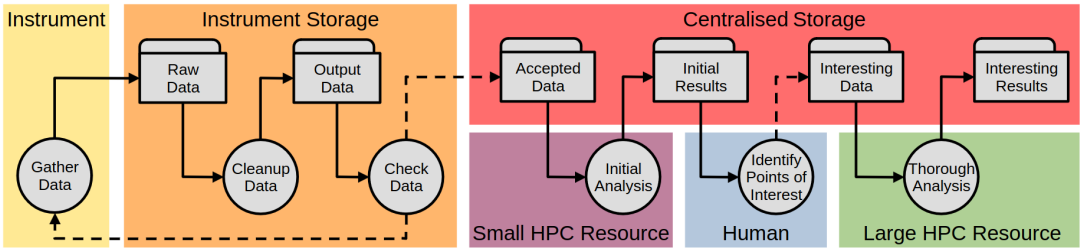

Figure 3: An example of a heterogeneous workflow

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, they create a directory containing the script defined by the recipe. The location of the directory is then sent to the runner, to be added to the job queue.

Conductors manage the jobs queue. They execute the jobs in the locations specified 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.

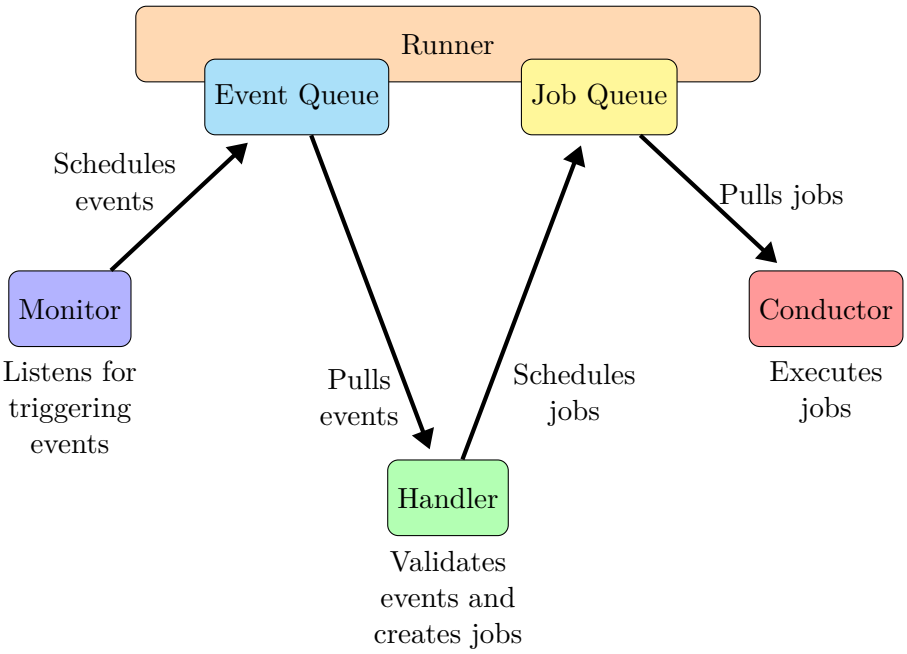

Figure 4: How the elements of MEOW interact

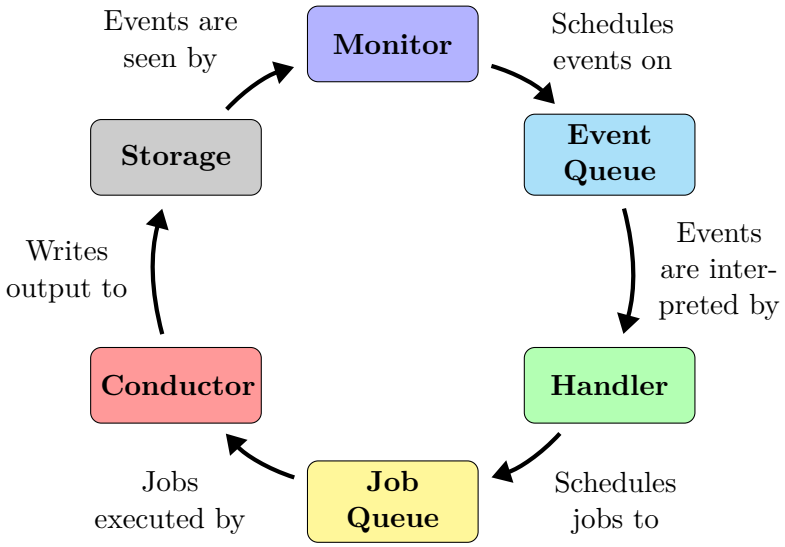

Figure 5: The cycle of MEOW's file events

2.2.2 The `meow_base` codebase

`meow_base`[5] 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. File events have the type `"watchdog"`, since the files are monitored using the `watchdog` python module.
  - `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.
- **Event patterns** inherit from the `BasePattern` class. An instance of an event pattern class describes a specific trigger a monitor should be looking for.
- **Monitors** inherit from the `BaseMonitor` class. They listen for set triggers (defined by given event patterns), and create events when those triggers happen. The file event monitor uses the `Watchdog` module to monitor given directories for changes. The Watchdog monitor is initialized with an instance of the `WatchdogEventHandler` class to handle the watchdog events. 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 serve primarily as a repository for the specific details of a given recipe. This typically includes identifying the particular script to be executed, but also contain validation checks of these instructions. The contained data and procedures in a recipe collectively describe the distinct actions to be taken when a corresponding job is executed.
- **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 repeatedly 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 repeatedly 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, for example, allows for ssl-encrypted communication, which may be a requirement in workflows with sensitive data. However, implementing network triggers using exclusively the `socket` library will provide MEOW with a fundamental implementation of network events, which can later be expanded or improved with other features (see section *4.3.2*).

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

Code available here: [2]

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. The list of patterns is pulled when starting the monitor, so patterns added in runtime are included. 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. When matching an event with a rule, all rules are considered, so if multiple rules use the same triggering port, they will all be triggered.

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 as a stream to a temporary file, in chunks of 2048 bytes. The temp files are created using the built-in `tempfile` library, and are placed in the os's default directory for temporary files. The library is used to accommodate different operating systems, as well as to ensure the files have unique names. When the monitor is stopped, all generated temporary files will be removed.

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 2048 bytes at a time, 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 Data Type Agnosticism

An important aspect to consider in the functioning of the network monitor is its data type agnosticism: the `NetworkMonitor` does not impose restrictions or perform checks on the type of incoming data. While this approach enhances the speed and simplicity of the implementation, it also places a certain level of responsibility on the recipes that work with the incoming data. The recipes, being responsible for defining the actions taken upon execution of a job, must be designed with a full understanding of this versatility. They should incorporate necessary checks and handle potential inconsistencies or anomalies that might arise from diverse types of incoming data.

> Justify. The file events don’t check for errors. The system is resistant, so errors don’t really matter. Protocol specific monitors could check better.

3.4 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. The tests verify the following behavior:

- Instances of the `NetworkEventPattern` class can be initialized, and raise exceptions when given invalid parameters.
- Network events can be created, and they contain the expected information.
- Instances of `NetworkMonitor` can be created.
- A `NetworkMonitor` is able to receive data sent to a listener, write it to a file, and create a valid event.
- You can access, add, update, and remove the patterns and recipes associated with the `NetworkMonitor` at runtime.
- When adding, updating, or removing patterns or recipes during runtime, rules associated with those patterns ore recipes are updated accordingly.
- The `NetworkMonitor` only initializes listeners for patterns with associated rules, and rules updated during runtime are applied.

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 Results

4.1 Performance Tests

To assess the performance of the Network Monitor, I have implemented a number of performance tests. The tests were run on these machines:

Identifier	CPU	Cores	Clock speed	Memory
Laptop	Intel i5-8250U	4	1.6GHz	8GB
Desktop	Intel i7-7700K	4	4.2GHz	32GB

The tests are done in isolation, without a runner. The events are verified by pulling them from the monitor-to-runner pipeline directly. The timing starts after all monitors have been started, but immediately before sending the messages, and ends when all of the events have been received in the runner pipeline.

4.1.1 Single Listener

To assess how a single listener handles many events at once, I implemented a procedure where a single listener in the monitor was subjected to a varying number of events, ranging from 1 to 1,000. For each quantity of events, I sent n network events to the monitor and recorded the response time. To ensure reliability of the results and mitigate the effect of any outliers, each test was repeated 50 times.

Given the inherent variability in network communication and event handling, I noted considerable differences between the highest and lowest recorded times for each test. To provide a comprehensive view of the monitor's performance, I have included not only the mean response times, but also the minimum and maximum times observed for each set of 50 tests, as well as the standard deviation.

Event count	Minimum time		Maximum time		Mean time		Standard deviation
	Total	Per event	Total	Per event	Total	Per event	
Laptop							
1	0.62ms	0.62ms	24ms	24ms	2.5ms	2.5ms	3.7ms
10	6.7ms	0.67ms	4,000ms	400ms	200ms	20ms	630ms
100	44ms	0.44ms	10,000ms	100ms	1,200ms	12ms	1,700ms
1000	550ms	0.55ms	22,000ms	22ms	6,800ms	6.8ms	4,700ms
Desktop							
1							
10							
100							
1000							

Table 1: The results of the Single Listener performance tests.

Given the large amount of variability in the results, new performance tests were run, repeating each test more than 50 times (n on the table).

Event count	n	Minimum time		Maximum time		Mean time		Standard deviation
		Total	Per event	Total	Per event	Total	Per event	
Laptop								
1	1000	0.63ms	0.63ms	16.0ms	16.0ms	2.4ms	2.4ms	0.89ms
10	500							
100	250							
1000	100							
Desktop								
1	1000							
10	500							
100	250							
1000	100							

Table 2: The results of the second suite of Single Listener performance tests.

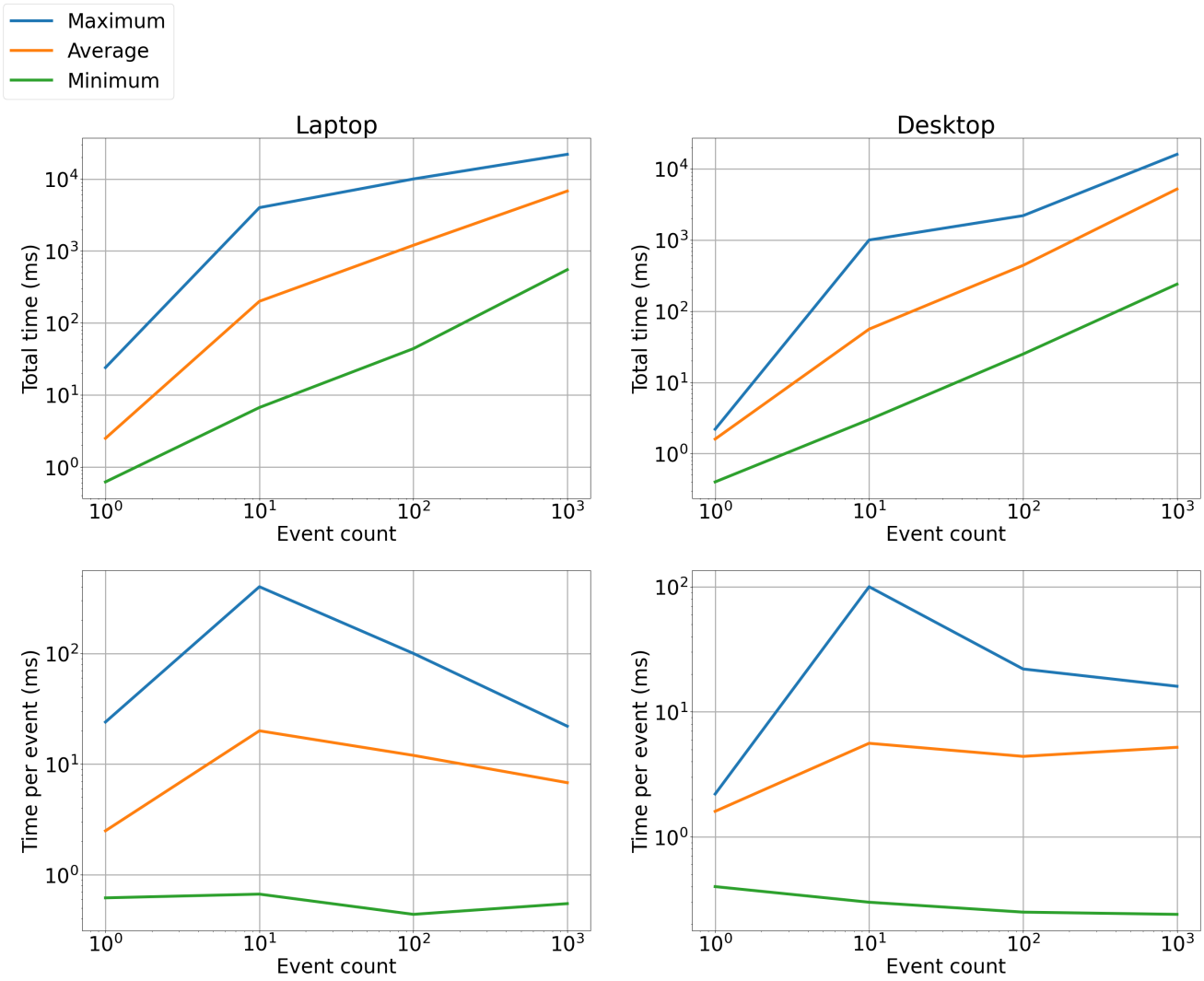

Figure 6: The results of the Single Listener performance test plotted logarithmically.

Upon examination of the results, an pattern emerges. The minimum recorded response times are consistently around 0.5ms per event for the laptop and 0.3ms per event for the desktop, regardless of the number of events sent. This time likely reflects an ideal scenario where events are registered seamlessly without any delays or issues within the pipeline, thereby showcasing the efficiency potential of the network event triggers in the MEOW system.

Conversely, the maximum and mean response times showed more variability. This fluctuation in response times may be attributed to various factors such as network latency, the internal processing load of the system, and the inherent unpredictability of concurrent event handling. It's worth noting that the standard deviation in these sets of data was consistently high. This suggests that the variability in the maximum and mean response times are due to high variability among the entire dataset, as opposed to singular outliers.

4.1.2 Multiple Listeners

The next performance test investigates how the introduction of multiple listeners affects the overall processing time. This test aims to understand the implications of distributing events across different listeners on system performance. Specifically, we're looking at how having multiple listeners in operation might impact the speed at which events are processed.

In this test, I will maintain a constant total of 1000 events, but they will be distributed evenly across varying numbers of listeners between 1 and 1000. By keeping the total number of events constant while altering the number of listeners, I aim to isolate the effect of multiple listeners on system performance. Once again, each test will be performed 50 times.

1000 was chosen as the total number of events to be sent due to its realistic representation of a high-load situation. While this number is higher than what I would typically expect the system to handle in a real-life application, it serves to provide a stress test for the system, revealing how it copes under an intensive load. This approach enables the identification of potential bottlenecks, inefficiencies, or points of failure under heavy demand.

A key expectation for this test is to observe if and how much the overall processing time increases as the number of listeners goes up. This would give insight into whether operating more listeners concurrently introduces additional overhead, thereby slowing down the process. The results of this test would then inform decisions about optimal listener numbers in different usage scenarios, potentially leading to performance improvements in MEOW's handling of network events.

Listener count	**Minimum time**	**Maximum time**	**Average time**
Laptop			
1	630ms	17,000ms	5,600ms
10	460ms	25,000ms	7,600ms
100	420ms	20,000ms	7,100ms
250	510ms	7,900ms	2,900ms
500	590ms	1,600ms	720ms
1000	920ms	3,200ms	1,500ms
Desktop			
1	240ms	16,000ms	5,200ms
10	240ms	19,000ms	4,000ms
100	250ms	10,000ms	1,000ms
250	270ms	12,000ms	900ms
500	310ms	330ms	310ms
1000	380ms	420ms	400ms

Table 3: The results of the Multiple Listeners performance tests with 2 significant digits.

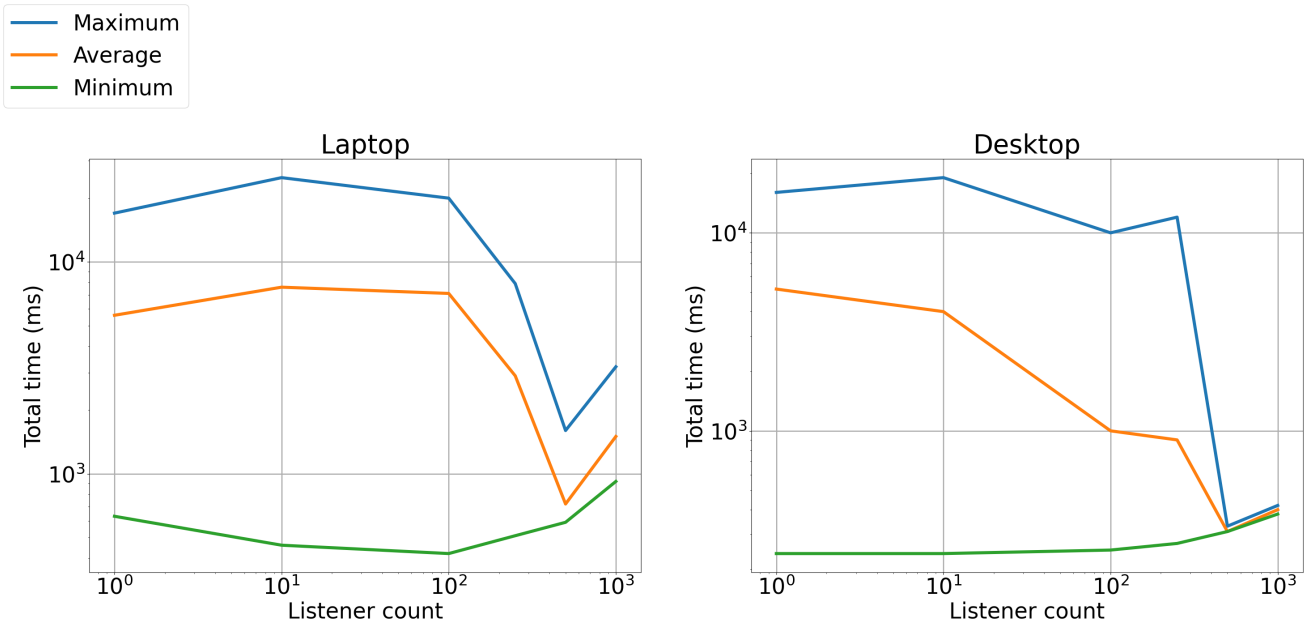

Figure 7: The results of the Multiple Listeners performance test plotted logarithmically.

The results of the Multiple Listener performance test provide fascinating insights into how the Network Monitor's performance scales with the number of listeners. From the data collected, I observe that there is relatively minor fluctuation, or a slight decrease, in maximum and average calculation time when distributing 1000 events across 1, 10, and 100 listeners. This implies that the system is able to handle increases in listener count up to a certain point without significantly impacting performance.

However, at 500 listeners, a noticeable drop in maximum and average calculation time occurs, followed by a slight increase when each of the 1000 listeners receives a single event. This trend could be attributed to the efficiency of the system in handling smaller, more distributed loads, possibly due to better utilization of threading.

Contrastingly, the minimum calculation time begins to increase once we reach 200 listeners, with further increases at 500 and 1000 listeners. This could suggest that while the system generally performs well under more distributed loads, the base overhead associated with managing multiple listeners starts to become more pronounced. Each listener requires some system resources to manage, so as the number of listeners increases, the minimum time necessary for processing might increase accordingly.

Therefore, the number of listeners initialized should be considered based on the expected traffic volume. This decision should balance the need for responsiveness against the capabilities of the system and its computational resources. For my tests, the overhead seemed to grow significantly once the amount of listeners passed 100, so the amount of concurrent listeners should likely not go above that.

4.1.3 Multiple monitors

The final test explores the performance of the system when multiple Network Event Monitors are run simultaneously. Although the current design and usage of MEOW wouldn't typically involve running multiple instances of the same monitor, it's important to anticipate potential future scenarios. Given the ever-evolving nature of computational workflows and the potential for different types of network event monitors to be developed, it's plausible to imagine a future situation where more than one network event monitor could be active at the same time.

In such cases, understanding the impact on system performance becomes crucial. This test helps evaluate how well the system handles the extra load and whether there are any unforeseen issues or bottlenecks when multiple monitors are active concurrently. This knowledge would be invaluable for any future improvements or enhancements in MEOW's design and implementation.

The test works similarly to the previous one, in that I will maintain 1000 events spread across a number of monitors. Each monitor will have 1 listener associated.

Monitor count	**Minimum time**	**Maximum time**	**Average time**
Laptop			
1	630ms	17,000ms	5,600ms
10	450ms	25,000ms	6,600ms
100	380ms	18,000ms	4,400ms
250	400ms	13,000ms	1,800ms
500	440ms	2,900ms	720ms
1000	520ms	2,300ms	700ms
Desktop			
1	240ms	16,000ms	5,200ms
10	230ms	20,000ms	6,500ms
100	240ms	18,000ms	2,900ms
250	250ms	7,600ms	800ms
500	260ms	300ms	270ms
1000	290ms	300ms	290ms

Table 4: The results of the Multiple Listeners performance tests with 2 significant digits.

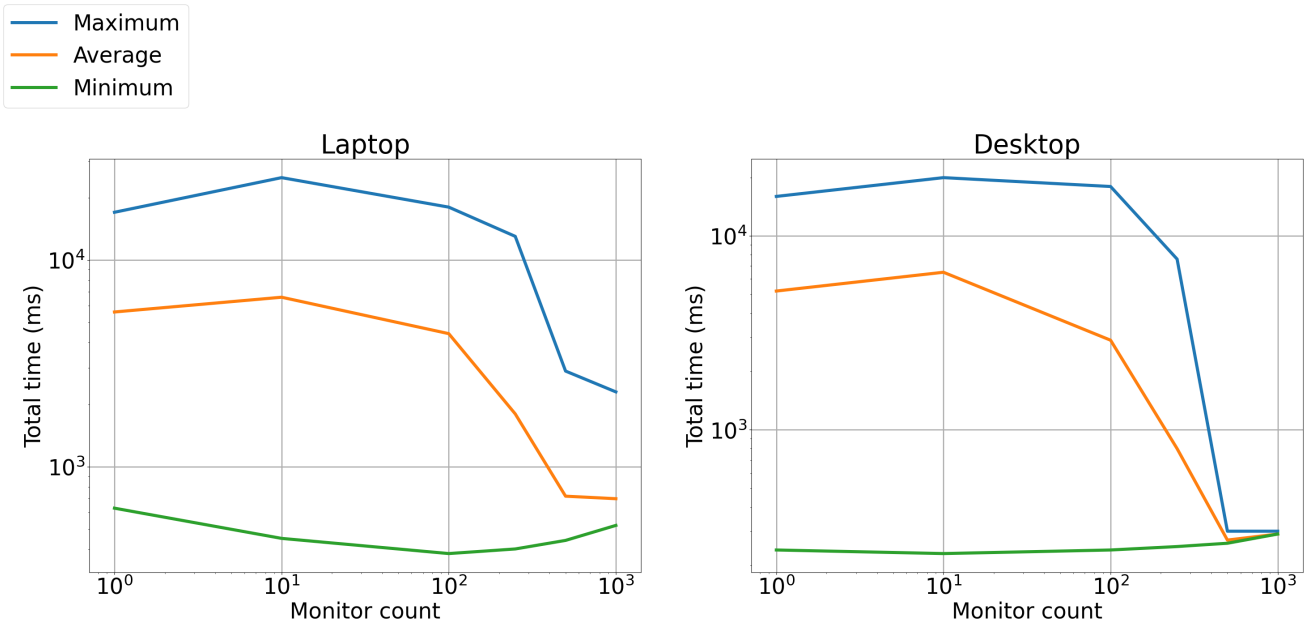

Figure 8: The results of the Multiple Monitors performance test plotted logarithmically.

The results are similar to the results of the previous performance test: the computation time drops significantly as the amount of events sent to each listener approaches 1. As with the previous test, the results show that having many threads doing less work is more efficient than a single thread doing all the work, with a small price in computational overhead.

4.2 Discussion

At the outset of this project, the primary objective was to integrate network event triggers into MEOW, enriching its capability to respond dynamically to network communications. I intended to design a solution that seamlessly integrates with the existing MEOW infrastructure and preserves the loose coupling between patterns and recipes.

Reflecting on this work, it is apparent that I have made significant strides in achieving this aim. The implemented network monitor successfully listens to network traffic and triggers corresponding jobs upon detecting matching patterns. This has expanded the capability of MEOW, enabling it to react and adapt to dynamic network events in real-time.

I ensured the solution would be a good fit within the existing codebase, leveraging existing structures where possible and introducing new components only when necessary. For instance, the network event patterns were designed with a similar structure to the file event patterns, ensuring compatibility with the current system. Furthermore, the data received by the network monitor was written to a temporary file, making the most of the infrastructure already in place for file events. This approach has been effective in achieving a seamless integration, preserving the flexibility of MEOW and allowing existing workflows to continue functioning without modifications.

Finally, the performance tests conducted provide a reasonable level of assurance that the network event triggers can handle a significant number of events, demonstrating the practicality and scalability of the solution. The network monitor has proven its ability to handle many simultaneous events, and the introduction of multiple listeners did not significantly affect performance, attesting to the robustness and scalability of the implementation.

Given these achievements, it can be confidently said that the project has met and even exceeded its initial objectives, thereby making a valuable contribution to the further development and versatility of the MEOW system.

4.3 Future Work

4.3.1 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.

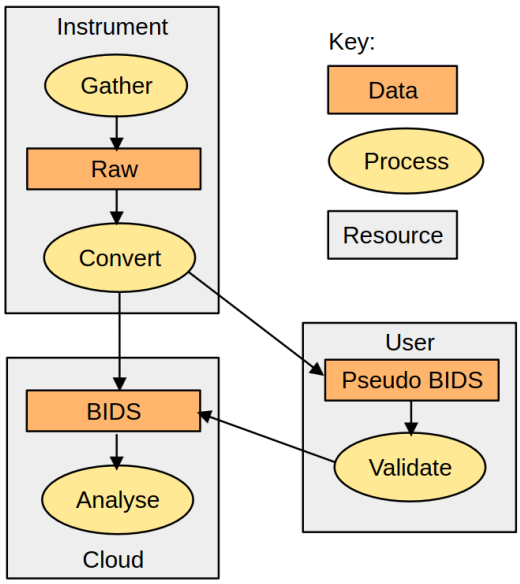

Figure 9: The structure of the BIDS workflow. Data is transferred to user, and to the cloud.

To illustrate the potential applications of network events in MEOW, I implemented a simplified workflow that involves two runners operating concurrently. These runners are initiated with almost identical, mirrored, parameters.

On receiving a network event, each runner is configured to respond by transmitting a network event to its counterpart. This simple setup mirrors the dynamic interaction of components in more complex, real-life workflows. It shows how the introduction of network events can enable the construction of workflows that require elements to communicate and react to each other's status.

Although this setup is quite rudimentary, it provides a tangible demonstration of the capabilities unlocked by the inclusion of network events. Using this as a foundation, it's easy to see how more complex arrangements could be built to accommodate more sophisticated workflows. In the context of the BIDS workflow discussed earlier, for example, the intercommunication between runners could represent the transfer and validation of data between different stages of the workflow.

4.3.2 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

I have successfully implemented a network event monitor into the Managing Event Oriented Workflows (MEOW) system. This new feature allows MEOW to handle network events and dynamically respond to data transmitted over network connections. The implementation was designed with modularity in mind, leading to a robust system that not only efficiently handles a multitude of events but also paves the way for future enhancements and extensibility.

The performance of the implementation was tested, providing insights into its strengths and potential areas for optimization. The results have shown that the sys-

tem can handle simultaneous events reliably, even in situations with multiple listeners or monitors.

References

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

[2] Nikolaj Gade. *meow_base*. https://git.ingemanngade.net/NikolajDanger/meow_base. 2023.

[3] David Marchant. *Events as a Basis for Workflow Scheduling*. https://sid.erda.dk/share_redirect/CA1fbrNHoD.

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

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