Autonomic Farm

Domenico Ferraro — d.ferraro7@studenti.unipi.it — 559813

Contents

1 Introduction

This report presents the design and implementation of an **Autonomic Farm**, a particular farm designed to efficiently handle dynamic workloads. A crucial aspect of AutonomicFarm's adaptability is its ability to monitor and gather metrics in real-time, such as global and local throughput, service time and arrival time. This report will present the logic behind determining the optimal number of workers by considering factors like arrival time, service time, and their impact on overall system. The AutonomicFarm was also implemented using FastFlow, and this report will highlight the differences in design compared to the native threads version. To assess the AutonomicFarm's performance, a series of synthetic benchmarks are presented and results of multiple experiments are commented. These include scenarios where the AutonomicFarm targets specific service times, adjusts to variable arrival times and task service times, and dynamically determines the ideal number of workers. This report presents and describes the details about all the different types of farms developed:

- **Farm**: A default and simple farm.
- **MonitoredFarm**: An extension of Farm, able to gather performance metrics at run-time.
- **AutonomicFarm**: MonitoredFarm's extension which pauses/resumes workers according to metrics data.
- **FFAutonomicFarm**: An autonomic farm implemented with FastFlow.

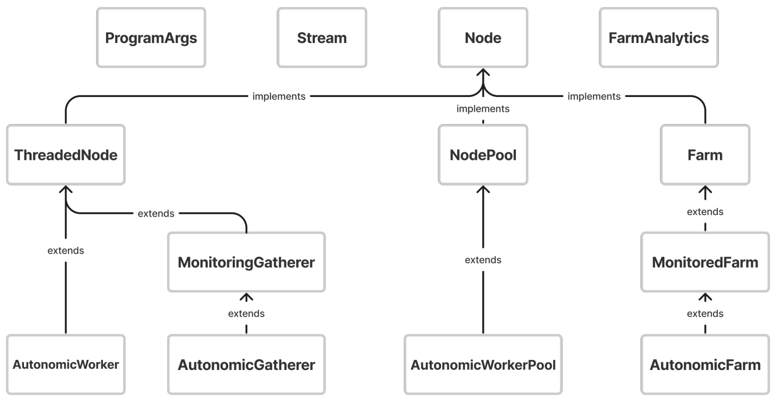

Figure 1: Project's class diagram

Here's an overview of the key directories and files within the project:

- **benchmark**: This directory houses benchmark programs designed to evaluate the performance and capabilities of every type of farm developed.
- **csv**: The `csv` directory is reserved for storing CSV files resulting from the benchmarks.

- **extern**: The `extern` directory is dedicated to store FastFlow.
- **include**: The `include` directory contains the source code.
- **CMakeLists.txt**: The CMake configuration file provides instructions for building all the benchmark programs made for every farm developed in this project. It specifies dependencies, and build-related details.
- **exe**: The `exe` directory holds executable files generated during the build process.
- **notebook**: This directory contains a Jupyter Notebook, provided to perform data analysis, visualization, and experimentation of any benchmark run.
- **tests**: The `tests` directory is dedicated to unit tests.

Multiple benchmark programs are provided and can be built via `cmake -H. -Bbuild` and then doing `make` in the `build/` directory. The following are the executables generated into the `exe/` directory: `farm` (MonitoredFarm), `autonomicfarm` (AutonomicFarm), `fffarm` (MonitoredFarm with FastFlow) and `ffautonomicfarm` (AutonomicFarm with FastFlow). They accept the following options:

`-w arg`	Number of workers (default: 4)
`-minw arg`	Minimum number of workers (default: 2)
`-maxw arg`	Maximum number of workers (default: 32)
`--stream arg`	Number of input tasks in the stream (default: 300)
`--service arg`	Service time values for tasks (space-separated) (default: 8 ms)
`--arrival arg`	Arrival time values for tasks (space-separated) (default: 5 ms)
`--target arg`	Target service time (default: None)
`--help`	Print the usage text

Table 1: Synthetic benchmarks options

For example, `--arrival 8 4 8` means that the first 33% of tasks arrive every 8ms, the second 33% of tasks arrive every 4ms and the remaining 34% of tasks arrive every 8 ms. Moreover, `--service 16 24` means that the first 50% of tasks are processed in 16 ms each, while the second 50% of tasks are processed in 24 ms each.

2 Implementation

2.1 Foundational elements

The main foundational element is the abstract concept of a **Node**. The various implementations of the farm parallel pattern provided and described by this report, and all the workers, emitters and gatherers, are concrete implementation of a Node. A Node, as conceptualized in this project, is an abstract entity akin to a node in a graph and designed to handle multiple inputs and execute a specified function on each input. Its flexible nature allows for the creation of diverse execution graphs, providing a powerful and adaptable framework for parallel processing. It simplified and made reliable the experiments and the farm designs which will be discussed.

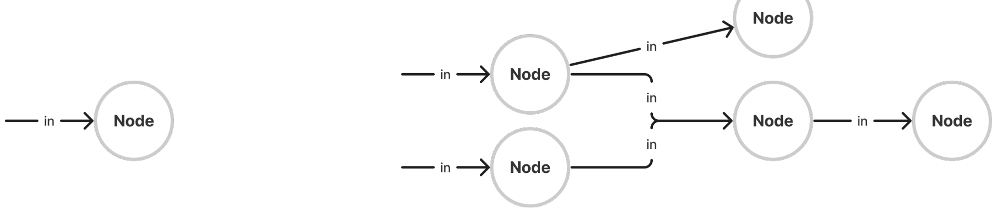

Figure 2: The Node abstract concept. On the left the simplest form of a node. On the right it is shown how simple it is to build a graph of nodes to represent a complex graph of parallel execution.

The primary responsibility of a Node is to execute a specified function on the input tasks it receives. The actual implementation of how the node stores input tasks and executes the function is left to the discretion of the implementer.

The Node interface provides a `run` method, allowing the node to start processing inputs and executing the specified function. This method initiates the running state, where the node actively consumes inputs and performs the designated operation. A crucial aspect of the Node's lifecycle is the ability to signal the end of the stream. The node exposes a method, the `notify_eos` method, that allows external entities to notify the node when no more inputs will be provided. The Node interface also includes a wait method, which is a blocking call. When invoked, the caller waits until the node reaches the end of its lifecycle which is reached when there are no more inputs, the node has completed its execution and the end of the stream was signaled.

Since a Farm is a concrete implementation of a Node, conceptually, this allows running and waiting for a farm, as well as attaching the farm output to other nodes in a pipeline manner or to another farm or to nothing else.

The **ThreadedNode** is the second main foundational element and it serves as a concrete implementation of the abstract Node concept. This specialized node associates a dedicated thread with its execution, allowing for parallel processing of tasks. A ThreadedNode continuously wait for input tasks from a Stream and executes the specified function on the obtained task. The **Stream** serves as an another foundational element. It is a FIFO buffer that maintains the order of tasks while providing thread-safe operations. The internal data structure of the Stream is a deque, ensuring O(1) access to the next task and O(1) send of a task.

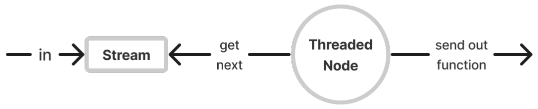

Figure 3: Visual representation of a ThreadedNode.

The Stream's next function is a blocking call that returns the next task from the stream when available. When the end of the stream is reached, it returns an empty optional to signal that the end of the stream was reached. When the Stream is empty, and the end of the stream is not signaled, the ThreadedNode will efficiently waits for the arrival of the next task. This mechanism ensures that the ThreadedNode remains idle until there is a task available for processing, preventing unnecessary resource consumption. The greatest advantage is that, if the ThreadedNode is not needed anymore, but it may be needed later, it is enough to stop sending to it any task since it will efficiently remain idle, eliminating the need to destroy and subsequently reconstruct the ThreadedNode. To achieve a high level of efficiency and minimize overhead, the Stream enforces zero-copy communication. The threaded nature of this node enables concurrent execution of multiple Nodes.

The last foundational element is the **NodePool**. It is a concrete implementation of a Node which aggregates multiple Nodes. Waiting and running a NodePool is implemented by waiting and running every internal Node. Signaling the end of the stream to a NodePool is implemented by signaling the end of the stream to every internal Node as well. Finally, sending a task to a NodePool is implemented by sending the task to one of its internal nodes, through round-robin scheduling.

2.2 Farm

The Farm serves as a base class that can be extended to implement various types of farms, scheduling policies, and internal behaviors. A Farm is made of two main components: a NodePool and another Gatherer Node. Gatherer's goal is to take in input the outputs of each node in the NodePool and to send out from the Farm. This paragraph provides details about the default implementation of a Farm, without any kind of extension. The NodePool is responsible for managing a collection of worker nodes. In the default implementation, each worker in the NodePool is represented by a ThreadedNode. The NodePool implements the round-robin scheduling policy. The Gatherer, also implemented as a ThreadedNode, serves as the node responsible for aggregating results from the worker nodes. Each worker send results to the Gatherer, which, in turn, executes the specified send-out function provided to the farm. Figure 4 provides a visual representation of a Farm.

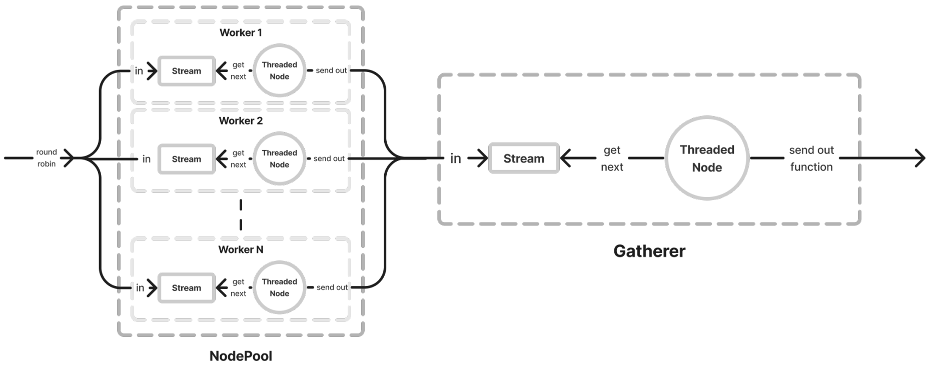

Figure 4: Visual representation of a Farm.

The default implementation does not involve any Emitter node. This design choice has the aim to lower the overheads to the minimum. The major advantage of this choice is to lower the number of processing elements needed by the Farm itself and the number of FIFO queues. The missing emitter would leave a processing element free to be used, for example, as another worker. Moreover, the external entities who send new tasks to the farm will call the NodePool's send function, so the round-robin scheduling policy is done by the external entities. This translates into a Farm Node that can be efficiently linked to another input Node by using the minimum number of processing elements: instead of having a previous Node which sends to Farm's emitter which in turn sends to the selected Node, we have a previous Node which sends to the Farm's worker by directly performing the round-robin scheduling policy. Information Hiding is heavily adopted by the Node abstract concept, which means that previous Node doesn't need to know that its next Node is a Farm to perform round-robin policy.

2.3 MonitoredFarm

The main point of the autonomic farm is to change its number of workers based on the knowledge about the arrival times and service times. Having that information as an input is not an option, since everything must be autonomic, so it is crucial to gain insights at run-time. Moreover, monitoring is needed to further analyze Farm's performance. The **MonitoredFarm** class extends the Farm class to gather information about the arrival time and the Farm's service time over time. The extension uses a **MonitoringGatherer** as a Gatherer node instead of the default one. The MonitoringGatherer node acts as the default Gatherer but also as a Monitor entity, monitoring the Farm to compute throughput and service time over time. After the execution of the MonitoredFarm, it is possible to obtain a **FarmAnalytics** object from it. This object consists of all the metrics obtained at run-time. Each metric is a pair $< value, time >$ where time is the number of milliseconds elapsed from the beginning of the Farm (when the Farm's run function was called). The following are the metrics collected at run-time:

- *farm_start_time*: point in time when the farm was started (i.e. its run function was called)
- *throughput_points*: raw throughput over time. It has multiple sources of noise, as discussed later
- *throughput*: the throughput over time without noise
- *service_time_points*: raw service time over time. It has multiple sources of noise, as discussed later
- *service_time*: the service time over time without noise
- *num_workers*: the number of active workers over time. An active worker is a worker not in idle, considered for the scheduling policy and busy working on input tasks
- *arrival_time*: arrival time over time. A value x means a new task arrived after x ms from *farm_start_time*

2.3.1 Computing the throughput

It is important to note that there are two types of throughput metric: the global and the local one. The global throughput takes into account the number of tasks gathered from the beginning of the farm. The following figure showcases the global throughput over time. In this example we have 4 workers, 500 input tasks, the arrival time is constant (a new task every 2 ms), the first 250 tasks will require 16 ms each to be processed and the last 250 tasks will require 32 ms each instead.

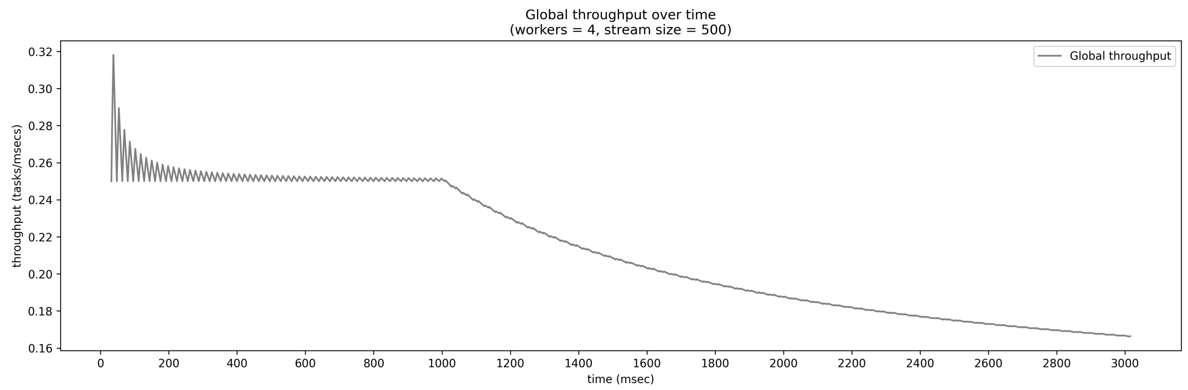

Figure 5: Global throughput. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the throughput value. This example can be run through `exe/farm -w 4 –stream 500 –service 16 32 –arrival 2` .

Since the first 250 tasks will require a computation of 16ms each and we have 4 workers, we expect that at around 1000ms from the start of the Farm, the tasks with higher service time are taken by the workers. As the plot shows, at around 1000ms from the beginning of the farm computation, the throughput gets lower since the workers need double the time. However, the change in throughput is not sudden and it requires a lot of time to understand which was the throughput at time 1000 ms. Since the worker's service time doubled, we expect the throughput to halve, but the global throughput is not providing such information. Because of that, at a given point in time the global throughput is not the actual throughput at that point in time since that metric is not so much sensible to changes. The local throughput, instead, provides the throughput value at a given point in time. The MonitoringGatherer computes it using a sliding window approach. At a point in time x, the window contains the number of tasks gathered, from 300 milliseconds before x to x. The throughput at time x is then computed considering how many tasks were gathered between the time frame $[x - 300, x]$. The following figure showcases the local throughput over time:

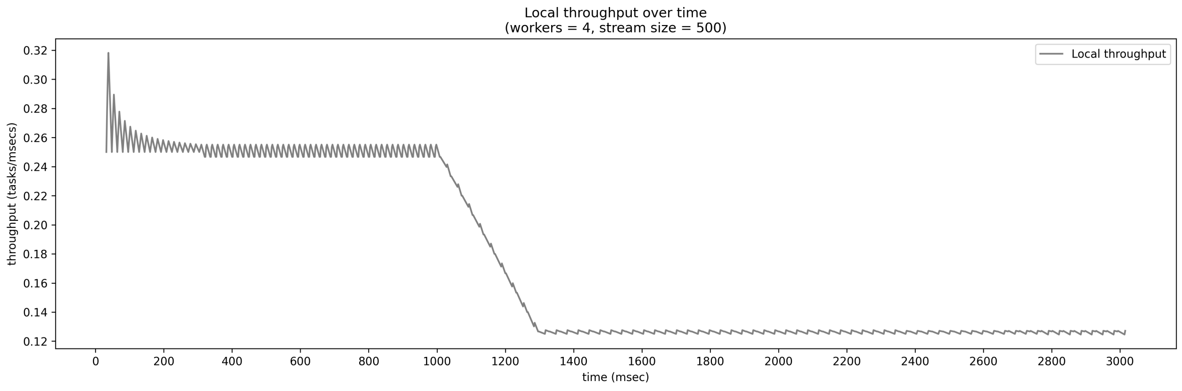

Figure 6: Local throughput. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the throughput value.

Note that the change in local throughput is sudden when the Farm starts taking from its input the tasks that require higher service time: at around 1300ms the throughput value is half the value at time 1000ms, as expected.

The computed throughput value, however, is not a smooth function. It is impossible to use it to understand the throughput slope or even the actual value and we can consider it as raw: since it is a function that quickly goes up and down, instead of knowing the exact throughput value, we know its range. To provide a much precise estimation, the MonitoringGatherer processes this metric at run-time and provides a final smoothed value. The following figure showcases in grey the raw throughput and in black the smoothed one, given the previous example.

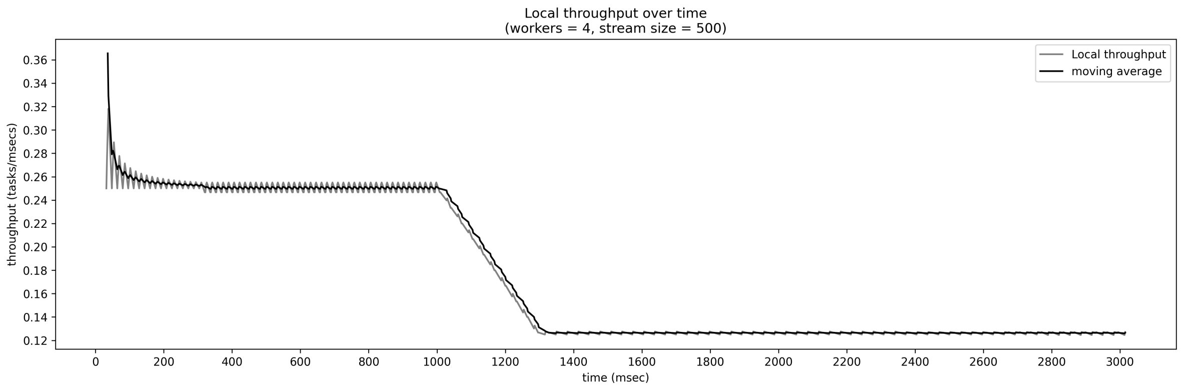

Figure 7: A much more precise estimation of the throughput over time. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the throughput value.

The MonitoringGatherer is able to provide a much more meaningful estimation of the throughput at each point in time: at each point in time, it uses a window of the last 5 raw local throughput values to compute the **moving average**. The computation doesn't involve looping through the whole window each time, instead the previous values are cached and reused at the next window.

2.3.2 Computing the service time

To compute the service time, the MonitoringGatherer leverages on the local smoothed throughput value. At each point in time the service time is equal to $1/throughput$. Since the throughput considered is the local throughput value, the service time is also the local one. Given the previous example, the following figure showcases the raw service time in grey and the final service time in black:

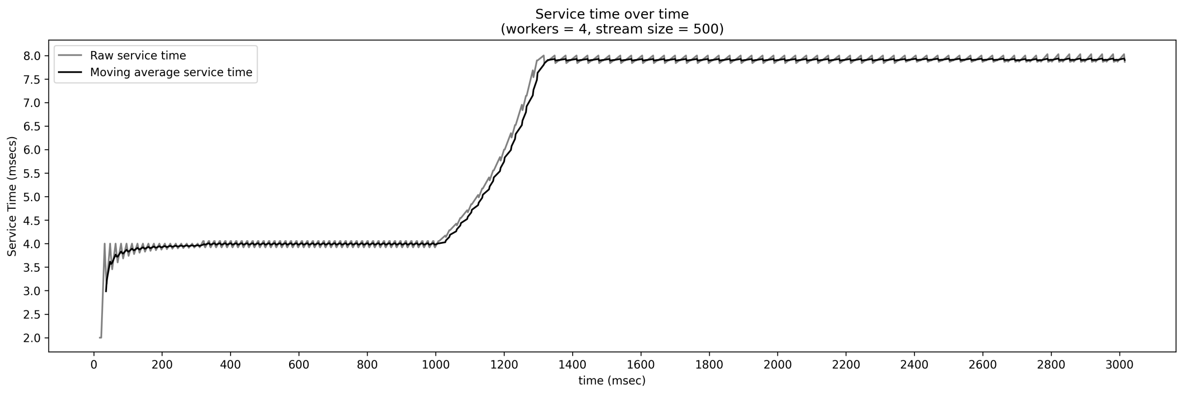

Figure 8: The Farm's service time. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the service time value.

As the figure shows, since there are 4 workers and the first 250 tasks require 16ms of computation, from the beginning to around 1000ms, the MonitoringGatherer computes a Farm's service time of 4ms. That value grows to 8ms when the Farm starts processing the input tasks that require 32 ms each.

Note that the goal is to compute a precise metric related to the farm's service time without leveraging on the knowledge of each worker's service time. It is a design choice with the goal to avoid adding any overhead to each worker, since that would require some sort of overhead to compute the timing and communicate it with the Gatherer. Moreover, if the service time is not as fixed as the given example, the service time of each worker wouldn't be the same and it would be very hard to have a reliable estimation of the actual Farm's service time. The following figure showcases how the default Farm behaves when given 4 workers, a stream of 500 tasks, when a new task arrives every 2ms, but the processing time required by each tasks is a random value between 8 ms and 32 ms.

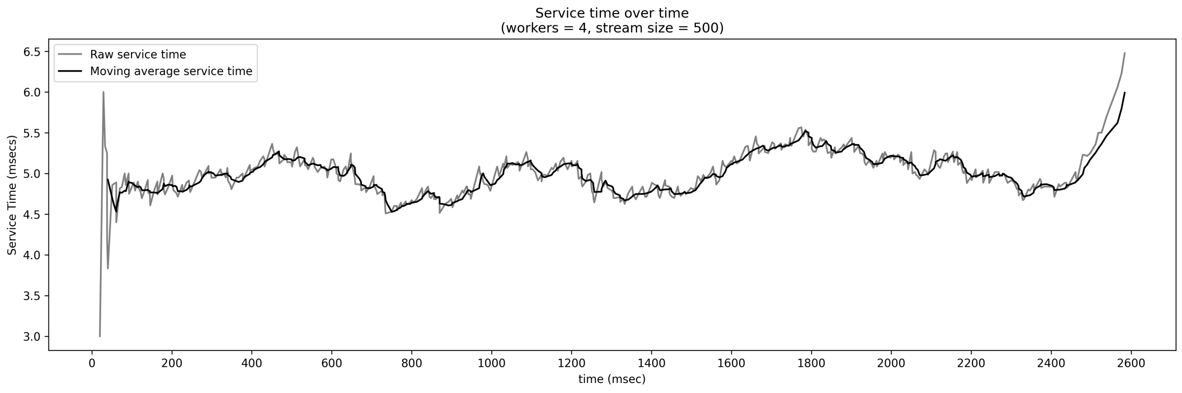

Figure 9: The Farm's service time when the input tasks have a random processing time. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the service time value.

2.4 AutonomicFarm

The **AutonomicFarm** extends the MonitoringFarm, and changes at run-time its number of active workers and its scheduling by leveraging on the monitoring data computed by the MonitoringFarm. Indeed, the Gatherer is an instantiation of an **AutonomicGatherer** which extends the MonitoringGatherer. This extension is about understanding at run-time whether the service time changed and pausing or resuming some workers according to the arrival time and the actual service time. The internal NodePool is an **AutonomicWorkerPool** which extends the NodePool by providing functionalities such as pausing and resuming the workers, and computing the arrival time. A worker of the AutonomicWorkerPool is an **AutonomicWorker** which is an extension of a ThreadedNode. The extension consists of the worker ability of being paused or resumed by external entities. When paused, the worker remains idle, waiting on a condition variable. The design of the AutonomicFarm is different since it implements some sort of auto-scheduling instead of round-robin scheduling. It means that the AutonomicWorker does not get its tasks from its input stream, but instead, it pulls new tasks on-demand from the main input stream.

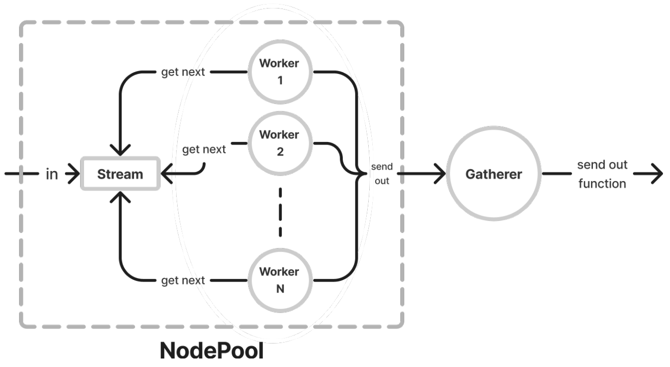

Figure 10: Visual representation of an Autonomic Farm.

The reasons behind the auto-scheduling design are the following. Since the workers have to be paused, and since there isn't an emitter Node, if we perform round-robin on every new task, we end up adding many tasks to Nodes that will be paused. If that happens, we could have two options:

- The worker pauses if and only if it finishes processing all its input tasks. However, it isn't a valid option since the autonomic behaviour wouldn't be as much instantaneous as possible.
- The worker pauses immediately, but there is an emitter Node which steals all the worker's remaining tasks and reschedules them. However, stealing all the remaining tasks and performing again round-robin scheduling takes time. During that time, the emitter Node must stop emitting. Moreover, we must waste a processing element to introduce an emitter Node. In addition to that, an emitter Node would have its input stream, so the external entities would send the tasks to the emitter, which will move tasks from its stream to another worker's stream. So many concurrent computations would introduce overhead that can be avoided by directly sending new tasks to the proper worker. Moreover, the AutonomicFarm would behave as a normal MonitoringFarm mainly, since the number of workers may not change so frequently.

The usage of auto-scheduling, instead, has important advantages:

- There isn't an emitter Node, so we have a processing element left to be used. The AutonomicFarm behaves as a normal MonitoringFarm when the number of workers is not changed
- Changing the number of workers is as simple as pausing or resuming workers and the overhead is kept as minimum as possible

Moreover, multiple experiments confirmed that the amount of overhead on having multiple readers on the main input stream is very low and it doesn't affect the whole performance. A reason could be that the overhead produced is in the order of nanoseconds, while the service times and the arrival times are in the order of milliseconds.

2.4.1 Variable number of workers

When the arrival time or the service time change, we may need to change the number of workers accordingly. The AutonomicGatherer computes the optimal number of workers, by dividing the worker's service time with the arrival time. However each worker may have a different service time, so it obtains a much more precise worker's service time value by multiplying the farm's service time with the current number of workers. Given the arrival time and the farm's service time at a given point in time, we can conclude that:

- If the farm's service time is higher than the arrival time, then the worker's service time over the number of workers is higher than the arrival time ($TS_{farm} > T_A => \frac{TS_{worker}}{num\_workers} > T_A$). In this case, the AutonomicGatherer computes the optimal number of workers by doing $\frac{TS_{worker}}{T_A}$, where $TS_{worker} = TS_{farm} * num\_workers$, and resumes the new workers (without going above the maximum).
- If the farm's service time is equal to the arrival time, we don't know if the worker's service time over the number of workers is lower than the arrival time, since the farm's service time is the maximum between that two variables. So it is not safe to compute the worker's service time from the farm's service time. To solve this problem, the AutonomicGatherer uses the first active worker's service time as a hint.

The advantage is that, since each worker's service time may be different, the farm's service time gives a much more precise metric and the workers are not spending any time computing their service time. The second advantage is that only one worker would spend some time computing its service time and that value is only used as a hint. It is safe to use only one worker's service time as a hint since the farm's service time value is dependent on that value. Even if we are in the unlikely situation when the first worker gets for the first time the first task with higher service time, we would easily recognize that situation since the worker's service time over the number of workers may be higher than the farm's service time. It is important to note that any service time or arrival time value would never be an exact value. Some sort of noise or error is always present. For this reason, the AutonomicGatherer takes into account a maximum error of 1.

After changing the number of workers, the farm's service time doesn't reach immediately the new desired value. This would hint the AutonomicGatherer to recompute again the number of workers needed. Recomputing again would involve looking at the farm's service time and the newer number of workers, leading the AutonomicFarm to come back to the previous number of workers or to make too many changes over time.

To overcome to that, after changing the number of workers, the AutonomicFarm targets a new farm's service time equal to the arrival time. After the change, the AutonomicGatherer won't recompute the number of workers again, but instead would consider the slope of the current farm's service time to understand if it is going to reach the target or not. If that's the case, the previous change in the number of workers is confirmed, otherwise a new number of workers is computed. This allows to avoid computing and changing the new number of workers too much in a short time frame, minimizing the overhead since that calculations and pausing or resuming the workers are done if and only if a change is actually needed. Another advantage is that this design helps to avoid any sudden change of mind in the number of workers. At the same time, if the arrival time or the farm's service time change, it is quickly recognized by looking at the slope. However, after 190 ms from the last time the AutonomicGatherer decided the number of workers, the target set is considered old so the AutonomicGatherer checks whether it is confirmed, otherwise a new target is computed.

To compute the farm's service time slope, the AutonomicGatherer performs rolling linear regression. It uses a moving window of the last 6 service time values and performs linear regression for each window, at run-time.

The following figure showcases the AutonomicFarm in action in a complex scenario. In this example there are 500 input tasks. The first 33% of them arrive every 8 ms, the second 33% of them arrive every 4 ms and the last 34% of them arrive every 8 ms. The service time of the first 50% of tasks is 16 ms while it is 24 ms for the remaining 50%. The initial number of workers is 4, which is also the maximum number of workers. The minimum number of workers is 1.

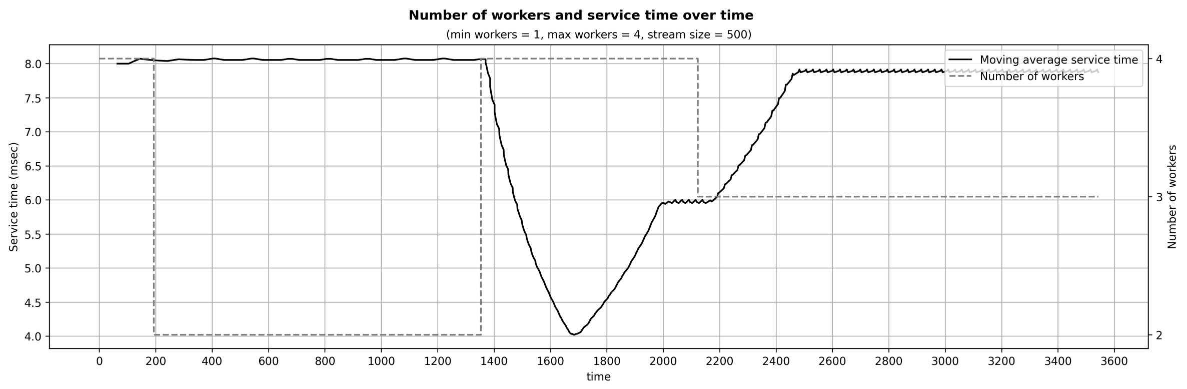

Figure 11: Service time and number of workers over time of an Autonomic Farm. On the X axis, the time elapsed from the start. On the Y axis the service time value and the number of workers (the dashed line). This example can be run via `exe/autonomicfarm -w 4 -minw 1 -maxw 4 –stream 500 –service 16 24 –arrival 8 4 8`.

The number of workers is decreased to 2 after 200 milliseconds since at that time we have new tasks every 8 ms and that tasks will be processed in 16 ms each. At around 1400 ms from the beginning, the AutonomicFarm recognizes that the farm's service time is higher than the arrival time, since the arrival time decreased to 4 ms. This leads the AutonomicFarm to resume 2 workers to increase its number of workers to 4. After the change, the farm's service time quickly decreases with a sudden slope: the AutonomicFarm immediately reacted to the new arrival time value. It will take around 250ms to reach the targeted 4 ms service time. However at that moment (around 1700 ms after the start) the newly arrived tasks will require a service time of 24 ms. Since the maximum number of workers is 4, the best thing that the farm can do is to keep the maximum number of workers: farm's service time would target the arrival time, which is 4 ms, but the best service time can be 6 ms. After around 2100 ms from the start, the arrival time increases to 8 ms, so the farm's service time increases from 6 ms to 8 ms. The AutonomicFarm recognizes that the increase of farm's service time is relative to the increase of the arrival time. The 4 workers are now too many and it pauses a worker to lower its workers number to 3.

3 Implementation with Fastflow

The AutonomicFarm implemented with FastFlow has a different design. It has an emitter entity which sends tasks to the workers and receives tasks from the external entities. When a task is received, the emitter puts it into an internal buffer. Each worker sends to the emitter a feedback when it has finished processing its task. When that happens, the emitter tracks that worker as available and will send to it a task from the buffer as soon as possible. Each worker, once processed a task, will send the result to a gatherer entity. The gatherer will send out the results and will send a feedback to the emitter to signal that a task was processed and shares the most updated service time. With this kind of feedbacks, the emitter has all the information to pause or resume workers, based on the same logic of the AutonomicFarm implementation using native threads.

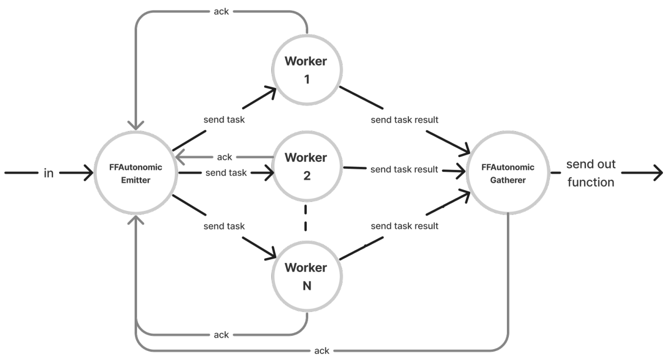

Figure 12: Visual representation of an autonomic farm implemented with FastFlow.

To pause a worker, the emitter sends a special task and the worker will wait with its condition variable, remaining idle. To resume a worker, the emitter will wake up that worker from its condition variable.

4 Benchmarks

This section provides some useful benchmarks of the AutonomicFarm implemented using native threads. No particular differences have been found between the native threads implementation and the FastFlow one, so the results of the AutonomicFarm implemented using native threads are presented as a representative of both.

4.1 Target service time

A metric used during development to evaluate the AutonomicFarm capabilities is the percentage of time the AutonomicFarm is able to stick with a target service time. It provides insights about the AutonomicFarm ability to compute the slope of the service time, to confirm or abort any previous decision. The following figure showcases an AutonomicFarm with 5000 input tasks that arrive every 2 ms, a minimum number of workers of 1 and a maximum and initial number of workers of 4. At run-time, the AutonomicFarm targets the arrival time, but in this benchmark it was forced to target a service time of 8 ms. The first 25% of tasks require a service time of 8 ms each, the second 25% require 32 ms each, the third 25% require 24ms each and the last 25% require 8 ms each. The target service time is successfully met 93.272% of the time. This example can be run via `exe/autonomicfarm -w 4 -minw 1 -maxw 4 –stream 5000 –service 8 32 24 8 –arrival 2 –target 8`.

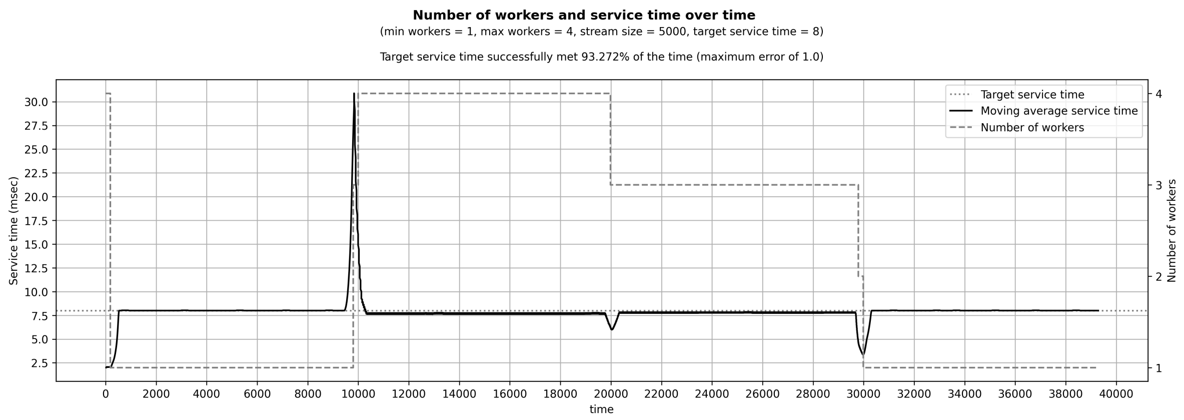

Figure 13: Service time and number of workers over time of an AutonomicFarm that targets a service time of 8ms. On the X axis, the time elapsed. On the Y axis the service time and the number of workers (dashed line).

4.2 Ideal service time

The following benchmark compares the AutonomicFarm's service time with the ideal service time considering a variable arrival time and task's service time. In this example the AutonomicFarm is not forced to target any service time, so it behaves as usual by targeting the best service time according to the input pressure and the tasks' service time. There are 5000 input tasks: the first 33% of them arrive every 8 ms, the second 33% of them arrive every 4 ms and the last 34% of them arrive every 8 ms. The service time of the first 50% of tasks is 16 ms while it is 24 ms for the remaining 50%. The initial number of workers is 4, which is also the maximum number of workers. The minimum number of workers is 1. This example can be run via `exe/autonomicfarm -w 4 -minw 1 -maxw 4 –stream 500 –service 16 24 –arrival 8 4 8`. The AutonomicFarm is targeting the right service time as the best service time (with a maximum error of 1 ms) and it is able to reach it as soon as possible by quickly pausing or resuming its workers.

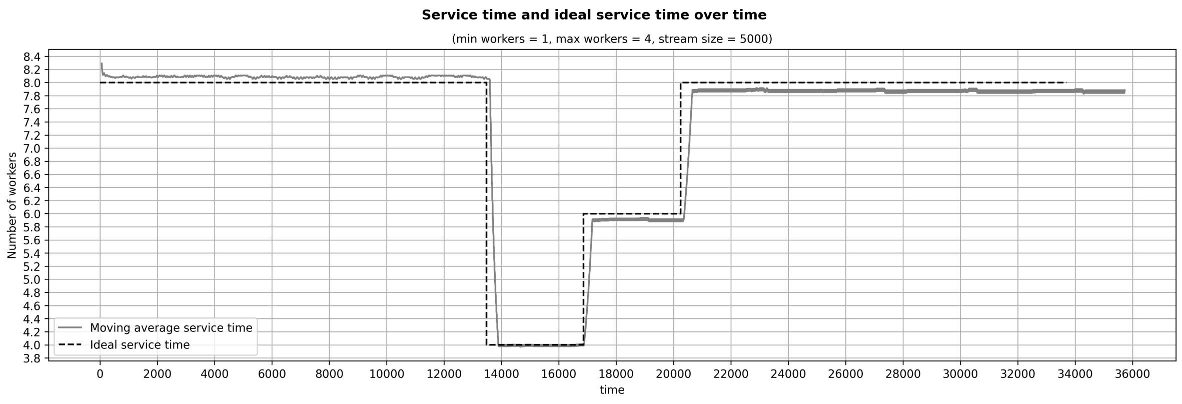

Figure 14: Service time and ideal service time over time of an AutonomicFarm with a variable input arrival time and tasks service time. On the X axis, the time elapsed from the beginning of the farm. On the Y axis the service time and the ideal service time (dashed line).

4.3 Ideal number of workers

The following figure showcases a benchmark with 500 input tasks that arrive every 8 ms. The first 33% of tasks require a service time of 16 ms each, the second 33% require 8 ms each, the last 34% require 32ms each. The figure compares the ideal number of workers with the actual number of workers over time. The AutonomicFarm successfully sets the best number of workers and reacts quickly to external changes. This example can be run via `exe/autonomicfarm -w 4 -minw 1 -maxw 4 –stream 500 –service 16 8 32 –arrival 8`.

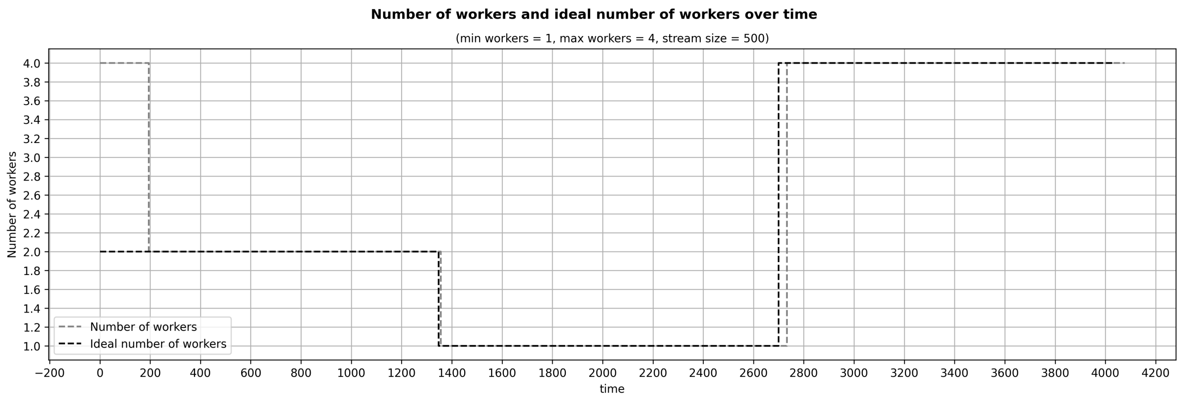

Figure 15: Number of workers and the ideal number of workers over time. On the X axis, the time elapsed. On the Y axis the number of workers set by the AutonomicFarm (dashed) and the ideal number of workers (black).