Scheduling Strategies for the Calvin IoT Environment

Sam Jabbar

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6065
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

The connection of devices to the internet is referred to as Internet of Things (IoT). By using IoT distributed network of devices will be able to communicate with each other. The management and forming of distributed applications is, however, very complex since IoT uses devices that have different capabilities and that do not have the same communication protocols. Calvin is an open source application environment developed by Ericsson that provides a distributed cloud for IoT. The distributed cloud can help to achieve low latency by using parallel processing nodes. Calvin is built upon the actor model, using the methodology dataflow programming. Using Calvin's programming model, the different parts of the system are represented by actors that are used to isolate and abstract functionality. Actors communicate with each other by message passing, however there is a network of runtimes that handle the actual dataflow. Several runtimes are applied in every device that will be used in IoT to get metadata about the capabilities of the devices and to get more generic and reusable actors [**1**].

In Calvin the scheduler is responsible for data transport and for triggering actor actions on in-data. The scheduler is necessary for execution of applications for each runtime. The current scheduler is a basic Non-Preemptive (NP) scheduler with limited knowledge because it only uses local information, such as what data has arrived from other runtimes, the rules for triggering an action, and the output data that will be sent forward to other actors whether they are local or remote. The aim of the master thesis is to improve Calvin's current scheduler. A new scheduling policy called Round Robin will be implemented and compared with the current Non-preemptive scheduling policy with respect to latency. A dynamic sorting part will be added to Calvin's scheduler and implemented for the two scheduling policies. The result of this thesis shows that the new strategies named New Non-Preemptive and New Round Robin are beneficial to use when most actors are not busy for every function call.

Keywords: Calvin, Internet of Things, Dataflow programming, distributed application, Actor model, scheduling, Round Robin, Non-preemptive.

Acknowledgements

I would like to dedicate a special gratitude to my supervisor at LTH Karl-Erik Årzén and my supervisors at Ericsson Research Per Persson and Harald Gustavsson for all their help and support during the whole process of work on this master's thesis.

Contents

1 Introduction

The internet has yet to reach its full potential and in the near future the 5G technology will be a reality. Therefore, a lot of companies are interested in connecting everything to the internet. The connection of devices to the internet is referred to as Internet of Things (IoT). By using IoT distributed network of devices will be able to communicate with each other. The management and forming of distributed applications are very complex since IoT uses devices that have different capabilities and that do not have the same communication protocols. This problem can be addressed by an application that is available for all distributed resources and is used as one environment.

Calvin is an open source application environment developed by Ericsson and provides a distributed cloud for IoT. The distributed cloud can help to achieve low latency by using parallel processing nodes. There are other frameworks developed for IoT, however, Calvin simplifies the work for the application developer. The runtime conceals a lot of the complex work and the application developer does not need to concern about things like actor migration or message passing [**1**]. Calvin is built upon the actor model, using the dataflow programming methodology. By using Calvin's programming model, the different parts of the system are represented by actors that are used to isolate and abstract functionality. Actors communicate with each other by message passing, however there is a network of runtimes that handle the actual dataflow. A runtime is applied in every device that will be used in IoT to get metadata about the capabilities of the devices and to get more generic and reusable actors.

In Calvin the scheduler is necessary for execution of applications for each runtime. It is responsible for data transport and for triggering actor actions on in-data. The scheduler that is used now is a basic Non-Preemptive (NP) scheduler that only uses local information such as what data has arrived from other runtimes, the rules for triggering an action, and the output data that will be send forward to other actors whether they are local or remote. The purpose of this master's thesis is to improve Calvin's scheduler. A strategy to improve Calvin's scheduler is for example to use non-local information such as information from the neighbor runtimes. Information from other runtimes can help to find the right actors to fire, without trying to fire all available actors. This will help to produce tokens in less time and therefore the problem with end-to-end latency can be reduced.

1.1 The Aim and Research Questions

The aim of the master thesis is to improve Calvin's current scheduler. Instead of trying to fire all available actors in the scheduler the new strategy will find and fire only actors that have the most probability to reduce latency. A new scheduling policy called Round Robin will be implemented in the Calvin Scheduler and a dynamic sorting part will be added to Calvin's scheduler and implemented for the scheduling policies (Non-Preemptive and Round Robin). All three strategies: Round Robin, Non-Preemptive with dynamic sorting part (New Non-Preemptive) and Round Robin with dynamic sorting part (New Round Robin) will be compared with the current scheduling policy (Non-Preemptive) with respect to latency.

1.2 A Sketch of an Approach

The approach of the thesis was to begin by studying Calvin and its internals, in particular the scheduler. The next step was to create a test scenario with constraints on application performance and design one or more applications for the test scenario. Instrumenting the code was essential to obtain necessary information. The requirements of the test scenario were addressed by devising a new scheduling strategy. The last stage was to compare the new scheduling strategy to the current scheduling strategy.

1.3 Delimitations

This thesis will focus on Calvin's scheduler and the other parts of Calvin will not be evaluated. Several scheduling policies could have been implemented to improve Calvin's scheduler, however, to adapt the work to the given time frame, in this thesis it has been chosen to only implement one scheduling policy (Round Robin). Furthermore a dynamic sorting part will be added. The improvement of Calvin's scheduler will be measured locally with respect to latency while, other changes due to the changed scheduling policy will not be considered. More over, the security of Calvin will not be mentioned in this thesis. During this thesis Calvin was constantly under development which also limited the work that was able to be done during the time period.

1.4 Outline of the Report

Methodological approaches are presented in chapter 2. The theory of internet of things and programming frameworks that are necessary to understand in the thesis are described in Chapter 3 and Chapter 4. Information about the scheduler is presented in Chapter 5. Chapter 6 contains the implementation of the new scheduler with comparison to the current scheduler in addition to possible future work.

2 Methodological Approaches

Six main methodological approaches have been used in this thesis: I. Literature review aimed at understanding internet of things, the structure and function of Calvin and the different parts of it (mainly the scheduler) and understanding programming and implementations in Python. II. Practical understanding of Calvin and its scheduler was important to detect a new strategy for improving the scheduler and to facilitate the implementation. III. Checking the code in the scheduler to make sure it works correctly and checking the behaviour of the strategy to assure it behaves as expected. IV. Calculate the time that was spent on different parts of the code to analyze the distribution of time and in this way finding possible errors. V. Testing of the new strategy to see if it fulfills the desired criteria. VI. Run the new schedulers and store the data for analyzing.

2.1 Theoretical understanding

In the beginning of the project the focus was on understanding the background to Calvin to get a clearer vision of the necessity of an improvement of Calvin's scheduler. Before starting to work with Calvin it was also necessary to understand the structure of Calvin, the different parts that it is built of and how they function. All this information was gained from Ericsson's webpage about Calvin and by literature search. The databases used for the literature search were primarily LOVISA and Google. Since Calvin implementation and actors are written in Python it was also essential to understand Python and Python libraries to be able to use it. The theoretical information about Python was collected by literature review and online tutorials.

2.2 Practical understanding

The next step was to collect practical comprehension to know how to use the theoretical knowledge. In order to understand Calvin,s practice was made on the tutorials and examples available in the open source. Simple applications were also built to apply the knowledge. After understanding how Calvin in total functions it was important to focus on the scheduler and obtain more detailed and advanced knowledge about it. Simulation of how the scheduler works and some simple exercises were made to understand Python and the main task of the scheduler.

2.3 Understanding the Scheduler using Logging and Print

After understanding the principle of the scheduler from the simulation it was time to work with the real Scheduler. Logging and print was the way to check what every method did in the Scheduler to make sure it worked correctly. During the development of new strategies for the scheduler logging and print was used to assure that the strategy behaved as expected. If the strategy did not behave as wanted it was rejected and a new strategy was developed.

2.4 Python Profiling

Sometimes the Scheduler and the Strategies behaved strange also when there were no syntax errors. Additionally, the error could not always be captured using Logging and print and then it was useful to use Python Profiling. With Python profiling the time that was spent on different parts of the code could be calculated which made it possible to know on what part of the code most time was spent. With this information the search area for the error could be reduced, for instance if the profiling shows that the most time was spent when the scheduler was sleeping then it was effective to focus on what makes the scheduling sleeping so much etc.

2.5 Calvin Tests

After a new strategy has been developed there are 999 tests in Calvin that the new strategy must pass through to see if it fulfills the desired criteria to function in Calvin. Going through the tests was the last and most difficult stage to pass. This was where the most ideas fail, and the next step is to either go through the idea and the code to try to correct it or if the idea is wrong the only way is to start all over again.

2.6 Run and store the data

When a new scheduler strategy passes through all the tests it was time to run the scheduler with the chosen scenario (See fig. 2.1). The data that is needed was collected by first running the new strategy with Round-Robin and then running the new strategy with Non-Preemptive scheduler policy. The time it takes for 5000 tokens to pass the actor *proc* was stored for every 1,2,5,10,20,30,40,50 and 100 applications for each scheduler. To analyze the data the sum and the mean value were calculated.

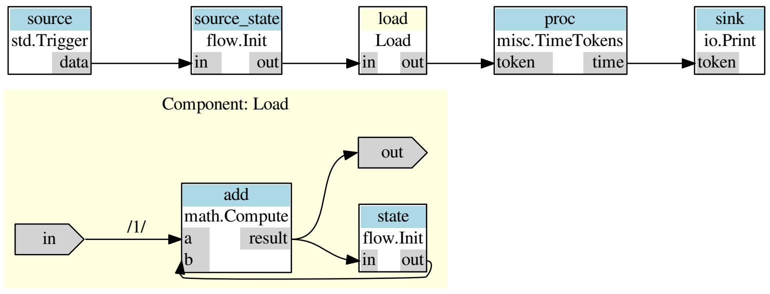

Figure 2.1 Test scenario with source of type std.Trigger.

3

Internet of Things

Internet of things (IoT) started in 1999 and refers to the connection of devices to the internet. The connection can be of both simple everyday devices such as a lamp or an alarm-system and more advanced devices such as a vehicle or an industrial robot [**2**]. With IoT very distributed network of devices will be able to communicate with each other and with humans. This increase of embedded systems in devices for integration of communication will lead to a larger general usage of the internet [**2**]. A crucial condition for the concept of IoT to work is that the devices must use a mutual method to connect to each other. The intention of IoT is to facilitate the interaction of devices in a secure way by using IT-infrastructure. With the use of IoT, devices will be easier to recognize and identify. Furthermore, with the ability to retrieve information from the internet the adaptive functionality of devices will be simplified [**3**].

IoT uses a lot of heterogeneous environments which means that the environments have different capabilities and use different communication protocols. This makes forming and handling distributed applications a very complex task to solve due to the applications developers need to consider both the platform and the communication protocols. It will be helpful if there was an application which reaches all distributed resources and is used as one environment. The developed applications must be able to use the available resources from other shared environments at the same time and that applies to the cloud infrastructure as well [**4**].

The possibility to execute code at different places simplifies the formulation of many real-world problems and makes it easy to implement the functionality of these problems [**1**]. Figure 2.1 shows how the different components of IoT are related and how IoT works. The IoT device, in this case an autonomous car and an industrial robot are connected to the internet. Data and analysis flow from and to the IoT device and to a remote control like for instance a tablet. The remote control is also connected to Wi-Fi. Data can also be sent to a storage device such as the cloud. A router enables the connections to Wi-Fi.

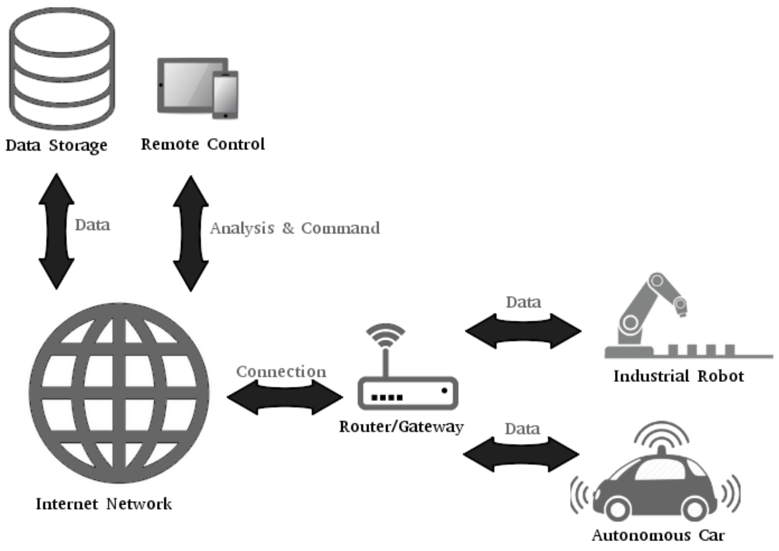

Figure 3.1 Internet of Things. A sketch of how the different components of IoT are related and how IoT works.

4

Programming Framework

Programming frameworks are platforms used to develop software applications. The purpose of programming frameworks is to be a foundation for building standardized code for applications or modules. Therefore, different problems in the development process can be decreased [**5**].

4.1 Actor Model

Calvin is an Actor based system. The Actor model is a model of computation which is designed to handle situations with high degree of parallelism, in particular to handle the concurrency problem. The Actor model was introduced by Hewitt et al in 1973 [**6**] [**7**]. There are other systems that use the Actor model such as CAL and the Ptolemy II project [**8**]. The paper "Native actors" describes the actor as an appropriate concept for multi-core processing [**9**].

4.2 Python

In Calvin, implementation and actors are written in the open source scripting language Python. Python is a very simple language with a very large library of add-on modules. Traditional compiled languages like for instance C and C++ tend to be faster than Python but they are not as easy to use as Python and therefore require more programming experience. These program languages need to use expensive libraries of software packages which makes it very difficult to scale for usage in small devices with limited storage capacity [**10**]. Python is beneficial for fast application development such as Calvin because it has dynamic typing and binding. Additionally, its high-level built in data structures make it suitable. Python is not complicated nor difficult to understand which makes it easy to use. [**11**].

4.2.1 Python Profilers

Sometimes it is very useful to measure the execution time of a particular part of the program. In Python the profile interfaces give a deterministic profiling of the program. There are three implementations of the profiling interface which the Python standard library offer: cProfile, Profile and Hotshot. Monitoring all function calls and returns, as well as all exception events, is called deterministic profiling. Furthermore, the intervals between the events have precise timings. For deterministic profiling in Python it is not necessary with instrumented code because during execution there is an interpreter. The interpreter adds overhead to the execution. The profiling adds much less overhead than the interpreter therefore this is less expensive. The statistic profiling can be used to identify bugs, inline-expansion, hot loops, high level errors in the selection of algorithms etc. There are some limitations, for instance the underlaying clock ticks at a rate of 0.001 s, which means that it takes some time before the profiler's call gets the time of the clock and the event will be captured. This will produce some errors due to the latency [**12**].

4.2.2 Python test

The reason to write test code for the original applications is to ensure that the code behaves as expected. It makes the development of the code easier, for instance adding more features. There is a type of errors that the developer can make that Python cannot catch during the compilation stage which can end up with the code behaving very strange without knowing where the error is. Therefore, the tests can be used once more to ensure that the code behaves as expected after the change. Also using tests is required nowadays more than before because of the security reasons. The defect code can for instance expose sensitive data or allow hackers to access private areas. [**13**]. In Calvin there are many tests that the new scheduler must pass to ensure that the new strategy will not affect the rest of Calvin negatively and to know if the new scheduler behaves as expected.

4.3 Distributed Cloud

A Calvin application consist of different parts (Actors) combined together. Calvin has an unique property that enables a distributed cloud for IoT, which means that some actors that are part of an application can be executed on different devices. This property allows the user to choose the most beneficial device to execute these particular parts of the application. This will be helpful to minimize the latency e.g. if the specific part migrates to another hardware which has much higher computing speed than the original one, or if the migrated part needs more computing power than the rest of the application then it will be beneficial to migrate for instance to the cloud. Hence, Calvin supports executing parts of its applications on different devices with different hardware and communication methods that use different protocols. This makes it possible for the application developer to focus on other things [**14**].

4.4 Calvin's internal parts

Calvin is an open source application environment that was designed by Ericsson to simplify development of IoT applications. Calvin's purpose is to simplify the application design by separating and isolating the functionality and the metadata. The benefit of using Calvin is that the Calvin runtimes scale well and can run on tiny devices as well as on full compute power available in the cloud. Calvin is built upon the actor model, using the dataflow programming methodology. By using Calvin's programming model, the different parts of the system are represented by actors and there is no distinction between for example cloud and device or server and client because they all share the same paradigm. The location of deployment in Calvin applications is not relevant if the right hardware is used. There is no need to change the code if an application is moved to another computer or device.

4.4.1 Runtimes

All the devices that are using Calvin will have Calvin runtimes executing in them. The purpose of runtimes is to get metadata about the capabilities of the devices and to get more generic and reusable actors. Calvin consists of both a development framework for application developers and a runtime environment for handling the running application.

4.4.2 Actors

Actors are used to isolate and abstract the functionality for instance by performing a computation or sensing a quantity. The runtimes execute dataflow applications by means of actors connected to a dataflow graph. Actors communicate with each other by message passing. An actor receives data objects called tokens from other actors to the in-port(s). The actor can also produce tokens to the out-port(s) to send out to other connected actors, however, the network of runtimes handles the actual dataflow. This will lead to applications adapting and scaling as required by duplicating actors or changing the location of another actor to another runtime. Every actor has inputs and outputs, the inputs and the outputs have each one a buffer with a limited size. Every in-port and out-port in an actor have an endpoint that connects two actors together when the applications run locally in the same device. For instance, an actor A is connected to actors B and C. The endpoints of actor A are found by manipulating the endpoints to get the actors B and C that are connected to actor A. Each actor has many first in first out (FIFO) in-and out-ports and it needs input data to produce the output. Calvin's actors can migrate between the run-times at the same time as the program is running to be able to write applications that will adjust to changing conditions [**1**].

4.4.3 Action

Executing an actor can be described as a chain of discrete stages that might lead to consuming tokens from the in-port and producing tokens to send out. Producing or consuming tokens change the internal state of the actor, this process is performed by an action. Only one action can be executed at a time [**15**]. Figure 4.1 shows a sketch of an actor with two actions. The actor has two inputs and two outputs. Tokens arrives to the inputs where they will be stored in FIFO buffers waiting to be processed. Once action *a* has finished then action *b* will be executed because action *b* has a lower priority than action *a*. The priority is assigned in action_priority as seen in Listing 4.1, for instance the action_priority = (a, b) which means that action *a* has higher priority than *b*. The produced tokens will be sent to the buffers in the out-ports.

```
from calvin.actor.actor import Actor, manage, condition, calvinsys,
    stateguard

class List_to_file(Actor):
    """
    Store data in a list then sends to a file after a number of
    data is stored in the list
    Input:
      token : data to write
    """
    @manage(["numbers"])
    def init(self):
        self.numbers=[]
    def write_to_file(self, x):
        wf=open("input.txt","a")
        for line in x:
            wf.write(str(line))
            wf.write("\n")
      wf.write("——————————————————————————————————————————")
        wf.write("\n")
        wf.close()
    @condition(action_input=['token'])
    def add_to_a_list(self, token):
        self.numbers.append(token)
        if (len(self.numbers) == 100000):
            self.write_to_file(self.numbers)
            self.numbers=[]
    action_priority = (add_to_a_list, )
```

Listing 4.1 *List_to_file*

4.4.4 Scheduler

The scheduler is responsible for data transport, and for triggering actor actions on in-data. It is necessary for execution of applications for each runtime. The scheduler which is used now is a simple Non-Preemptive scheduler with limited knowledge because it is only using local information such as what data has arrived from other runtimes, the rules for triggering an action, and the output data that will be sent forward to other actors whether they are local or remote.

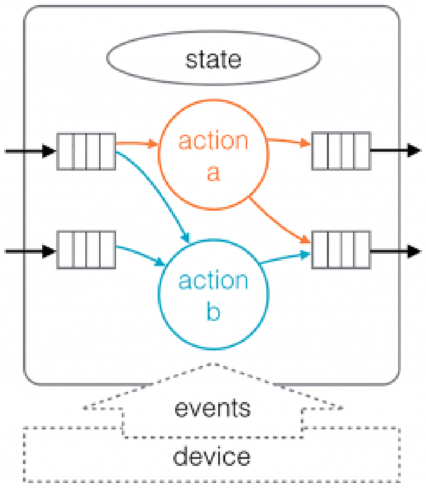

Figure 4.1 A sketch of an Actor with state and two actions[**16**]

4.4.5 Migration

In Calvin the actors communicate via the in and out ports which means there is no other way to influence the state of an actor. This property makes the actor isolated from the others in the application and that is the key to why the actors can be moved (migrated) separately to other runtimes without interrupting the application execution or making any changes to the functionality of the application [**14**].

There are two ways to initiate a migration, the first one is manually by sending a command using a Calvin Control API to the runtime that we want to migrate an actor from. The other possibility of migration is automatic migration of an actor/actors and that is done by the runtime itself to achieve the best performance [**16**]. There are many local or external factors that will trigger the runtime to migrate some parts of an executed application. It can be a high load on the runtime or deploying actors to the runtime such that it exceeds the capacity of the runtime to hold all actors together. An external factor can be a hardware or network change [**14**].

4.4.6 Capabilities and Requirements

The capabilities of Calvin are the properties and abilities of the runtime or of the device that the runtime is implemented on. The capabilities in Calvin can be stored either by a central repository or a secure distributed hash table (DHT). The decision of which of these two implementations to select depends on the configuration. Capabilities are information taken from the device that the runtime is running on. For instance, a camera that produces images. This information is considered as capability and up on that the information about location, ownership is also considered as a kind of capability.

For Calvin to work there are some requirements that must be fulfilled. The choice of actor will decide the requirements that must be fulfilled for the application since the actor has different requirements on the runtime. The requirements can be for example the presence of a timer, the ability to measure temperature, or access to a file system etc. The applications also have requirements on the properties like for instance a certain runtime or a specific location [**17**].

4.4.7 Calvin on Graph Level

A Calvin application consists of a number of actors that are connected to each other. The actors are already predefined and exist in the actor library, but it is easy to write a new actor if it is necessary, for instance to collaborate with new hardware or to include some features that are in some actor collection. By adding the new actor to the library, it becomes available to use by other actors and reuse it in other applications. An actor's task is doing a simple operation on the received data from one or more inputs and to write the results to one or more outputs. Connecting actors together leads to building complex applications that can perform complex tasks.

By running Calvin runtime on an IoT device concurrency problems can be easily handled since different applications are available on the same device at the same time. Parts of the applications can be distributed on different devices since Calvin supports distributed applications.

Executing Calvin on different types of devices means that Calvin has to deal with different communication types, protocols and features. At this level these different problems are hidden from the application developers and the different devices are treated as one type [**1**].

An example of a Calvin application on graph level is showed in Figure 4.2. The application contains a component called Load which is a group of standard actors that can be treated as one actor. The last actor is not a standard one, it is written to save results in a list and then after the saved results reach a specified number they are sent to a file. The application task in Figure 4.2 computes the time a token spends inside the component *Load*, by computing the time difference between two actors. One token is sent to *Load* and then to the actor *time_delta* and another actor is sent directly to *time_delta*. The results are temporary saved in a list and when the number of stored results reaches 100 000 they are sent to a file.

Figure 4.2 describes what every actor do in the application. The first actor is a standard actor called *Source*. *Source* is of type *std.Trigger* and it sends out tokens every 0.5 seconds. The tokens arrive to *Load* and *time*_0 at the same time. The actors *time*_0 and *time*_1 are of type *time.Timestamp* which receives a token and sends out a time stamp on seconds form. *Time_delta* is of type *math.Compute* which will compute the difference between the times-stamps from *time*_0 and *time*_1. The actor *Sink* will send every 100 000 results to a file. The component *Load* consists of two actors: *state* and *add*, that are connected with each other in a loop. *State* is of type *flow.Init* and has a pre-defined initial value as shown in the script in *Listing*1. As soon as the application is started the data, which will be zero, is sent to *add*. When *state* receives new data it will be sent back to the in-ports in *add*. *Add* is of type *math.Compute* and its only task is to add the tokens from the in-ports a and b. Listing 1 also contains the code for instantiating the actors and connecting the together.

```
component Load() in -> out {
    state : flow.Init(data=0)
    add : math.Compute(op="+")

    .in > /1/ add.a
    state.out > add.b
    add.result > state.in, .out
}

source : std.Trigger(tick=0.5, data=true)
time_0 : time.Timestamp()
time_1 : time.Timestamp()
load : Load()
time_delta : math.Compute(op="-")
sink : io.List_to_file()

source.data > time_0.trigger, load.in
load.out > time_1.trigger
time_1.timestamp > time_delta.a
time_0.timestamp > time_delta.b
time_delta.result > sink.token
```

Listing 4.2 *Application example*

4.4.8 Calvin on Actor Level

The actors are written in Python using the actor class from Calvin as a base to build an actor, from actor class imports python-decorators: *actor*, *manage*, *condition*, *calvinsys* and *stateguard*. In the example in Listing 4.2 only the imports: *actor*, *manage* and *condition* are necessary. *Actor* is needed as an argument in the class *List_to_file*(*Actor*). *Actor* is the base class of all actors and should always be included, while *manage*, *condition* and *stateguard* are python-decorators that are used to simplify development.

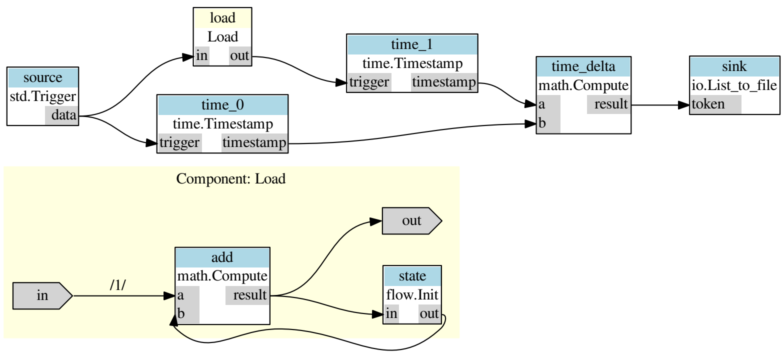

Figure 4.2 Calvin application shown on the graph level.

At migration attributes of the actor are automatically managed and the runtime gets information about the attributes by *manage*. In order for an action to be executed, tokens must be available for in-ports and there must be space available for tokens in out-ports. *Condition* shapes the ports that fulfill these requirements [**18**].

The code example for the actor *List_to_file* shown in Listing 4.2 consists of all the doc-string which includes the description of the tasks that the *actor* will do and the names of the in-ports and the out-ports. In this example there are no outputs and therefore it is not necessary to add any output description. *Manage* takes the numbers list as attribute. As shown in the code *Manage* uses the init method to inform the system, for instance which state of the actor must be serialized on migration [**18**]. The *Write_to_file* method is needed to open, write data and then to close the file. The next decorator is *condition* which specifies the needed input data and the required output space. If not both input and output conditions are fulfilled the action cannot be fired [**18**].

The *Add_to_a_list* method is used to append the new token to the list and check if the length of the list is 100 000 because then the list will be emptied and the *Write_to_file* method is called. *Action_priority* contains the conditions that are needed to make sure that the action with the highest priority will be ready to fire [**18**]. In this case the actor only contains one action.

Listing 4.3 shows the python code inside the standard actor *Trigger*. The actor has two parameters: *tick* and *data*. *Data* is produced every *tick* seconds. The *Trigger* has two actions *start_timer* and *trigger*. The *start_timer* action has higher priority than *trigger*. This means that the *start_timer* will first call the function *start* and then when it is finished the *trigger* will start. The state changes when *start* initializes a new timer.

stateguard decorator, comes before the *condition* decorator. The *stateguard* decorator improves the criteria for selection an action for instance depending on the internal state of the actor it decides whether to allow the action to run or not.[**18**].

```python
from calvin.actor.actor import Actor, manage, condition, stateguard

class Trigger(Actor):
    """
    Pass on given _data_ every _tick_ seconds
    Outputs:
        data: given data
    """

    @manage(['tick', 'data', 'started'])
    def init(self, tick, data):
        self.tick = tick
        self.data = data
        self.timer = None
        self.started = False
        self.setup()

    def setup(self):
        self.use('calvinsys.events.timer', shorthand='timer')

    def start(self):
        self.timer = self['timer'].repeat(self.tick)
        self.started = True

    def will_migrate(self):
        if self.timer:
            self.timer.cancel()

    def did_migrate(self):
        self.setup()
        if self.started:
            self.start()

    @stateguard(lambda self: not self.started)
    @condition([], ['data'])
    def start_timer(self):
        self.start()
        return (self.data, )

    @stateguard(lambda self: self.timer and self.timer.triggered)
    @condition([], ['data'])
    def trigger(self):
        self.timer.ack()
        return (self.data, )

    action_priority = (start_timer, trigger)
    requires = ['calvinsys.events.timer']
```

Listing 4.3 A standard actor Trigger [**19**]

4.5 Building Applications for the Test Scenario

For this project two scenarios have been used for analysis. The scenarios are very simple applications and consist of five components: source, source_state, load, proc and sink (See Figure 2.1).The source for the first scenario is of type *std.Trigger* and the source for the other scenario is of type *std.Counter*. The components work as follows:

1. Source:
Source is an actor of type *std.Trigger* or *std.Counter*. The *source* of type *std.Trigger* will send tokens every pre-decided time period [**19**]. For this project the pre-decided time is 0.5 seconds. The *source* of type *std.Counter* will send tokens with no time delay.

2. Source_state:
The tokens from the *source* will reach the *source_state* that is of the type *flow.Init* which is used to send an initial data [**19**]. The quantity of tokens that will be sent can be chosen by specifying the parameter nbr in the *flow.Init*. For instance, nbr = 1 for the Round Robin scheduling policy since only one token at a time is needed and nbr = 4 for the Non-Preemptive scheduling policy since 4 actors at a time are needed.

3. Load:
The tokens from the *source_state* will reach the *load* which is a component of actors. *Load* consists of two actors; *add* and *state*. *Add* is of the type *math.Compute* which will add data from a and b inports [**19**]. *State* is of the type *flow.Init* which is the same type as *source_state*. By connecting the actors as in Figure 2.1 a loop is created which will delay *add* from producing tokens.

4. Proc:
The token will continue from *load* to the actor *proc* which is of the type *misc.TimeTokens*. *Proc* will measure the time taken to consume N tokens. N is the number of tokens and in this case it was chosen to be 5000 tokens.

5. Sink:
Proc will send the data to *sink* which is of the type *io.Print*. *Proc* will print the total time of producing 5000 tokens for this scenario. The data is then analyzed at the end of the project (See figure 2.1).

```
component Load() in -> out {
  state : flow.Init(nbr=4, data=0)
  add : math.Compute(op="+")
  .in > /1/ add.a
  state.out > add.b
  add.result > state.in, .out
}
source : std.Trigger(tick=0.5, data=1)
source_state : flow.Init(nbr=4, data=0)
proc : misc.TimeTokens(N=5000)
load : Load()
sink : io.Print()
source.data > source_state.in
source_state.out > load.in
load.out > proc.token
proc.time > sink.token
```

Listing 4.4 *Scenario written in Calvin script*

4.6 Related Frameworks

4.6.1 Capecode

Capecode is a modeling environment for building IoT applications and code generators with a block diagram editor [**20**]. Composing heterogeneous devices and services in IoT is simplified by Accessors [**21**]. Capecode is a configuration of Ptolemy II. Ptolemy II is an open source software that supports actor-based systems. [**22**].

Capecode is built from the three components Nashorn, Ptolemy II and Vert.x. The first component, Nashorn is a JavaScript engine that provides full access to Java and runs on the Java virtual machine. JavaScript is a scripted, gradually-typed language and Java is a strongly-typed object-oriented language. By combining Java and JavaScript, Nashorn contributes with an environment that benefits programming. Additionally, the networking and I/O libraries of Java simplifies building interfaces to devices. The second component, Ptolemy II makes importing accessors available from the default library of actors or from other libraries (any index.json file included directory/website with a list of available accessors). In Ptolemy II composing accessors is done by Vergil which is a graphical block diagram. Vergil is also responsible for building swarmlets that makes it possible for mixture of accessors with usual Ptolemy II actors. The third component, Vert x is a library for event-driven programming.

If the function that is needed is not provided by an accessor or an actor in the Ptolemy II library an example of the JavaScript actor may be used to build an own actor or accessor. JavaScript actors are sometimes a better option than accessors like for instance if the functionality is not host-specific [**23**].

Usage of the actor model in CapeCode allows events like for instance scaling, migration and load balancing to be managed on a fine-grained level. This is possible by preventing the internal state from changing unless it is responding to an event or data [**17**].

4.6.2 AWS Lambda

Amazon Web Services Lambda (AWS Lambda) is a platform that manages IoT. AWS Lambda enables usage of third party library of own choice and supports multiple languages including Java, Node.js, *C*# and Python.

AWS Lambda runs the code in response to several events and is a server-less computing. After supplying the code, the administrations of the compute resources such as maintenance of server and operating system, automatic scaling and code monitoring are all automatically managed by AWS Lambda.

The code in Lambda are also known as Lambda functions and before they can run they must be triggered by an AWS resource like for instance Amazon Simple Storage Service (Amazon S3) buckets or Amazon DynamoDB tables. The lack of affinity to the infrastructure by the code makes it possible for AWS Lambda to quickly launch the necessary amount of copies of the code to measure the rate of incoming events. Compute to data can be added to AWS Lambda as it passes through the cloud due to extension of other AWS services with custom logic [**24**].

Several AWS Lambda functions can process the same events with Amazon Kinesis that receives data with events and store them for 24 hours. After processing the incoming events an AWS Lambda function provides low-latency access by storing the event data in a table in Amazon DynamoDB. DynamoDB changes a configuration value to enable the necessary table capacity to be provisioned. For devices to regain data from DynamoDB they need to call a synchronous interface that is allowed by another AWS Lambda function. A third AWS Lambda function provides cost effective and lasting archival by storing data in Amazon S3 which also facilitates the access of the data for analysis. Analysis of data is done by Amazon Elastic MapReduce (EMR) that runs the events from DynamoDB and S3 [**25**].

4.6.3 Node-RED and Distributed Node-RED

Node-RED is an IoT environment that is based on data-flow programming concept, which provides a browser-based flow editor. The application developers can easily build complex applications by using Nodes (In Calvin they are called Actors). The Node-RED Integrated Development Environment (IDE) was created first as open source at IBM 2013. JavaScript script language is used to write nodes for Node-RED platform [**26**]. JavaScript functions can be created in the editor using a text editor.The Node-Red runtime is built on Node.js [**27**].

Node.js is an event-driven I / O framework based on the JavaScript engine V8. It is created to write scalable network applications such as web servers. The programming language used is JavaScript that is executed on the server side. In node.js, almost no functions are performed that directly block I / O. This prevents deadlock from occurring. It is handled through callbacks, instead of waiting for a result from I / O, Node.js can execute another code in the meantime. When I / O is completed, the callback reference is called and the result is handled [**28**]. Node.js is a light weight framework which can run on low cost devices such as Raspberry Pi and other tiny devices that can be connected to the internet.

Node-RED uses the flow-based programming model. It is possible to quickly build applications in a simple way for instance by using a graphical drag and drop interface. This is possible due to the control of the data movement through an application and the well-defined interfaces between the components[**17**].

The distributed Node-RED extends the existing IoT data flow-based systems by creating a platform appropriate to execute on a range of run time environments, and supports data flows that can be partitioned manually. The distributed Node-RED supports automatic dynamic sorting and distribution of data flows based on participating resource capabilities and constraints required by the developer around cost, performance and security.

4.6.4 Main Differences from Calvin Platform

There are three main factors that makes Calvin different from the other platforms that exist today:

◦ Calvin allows greater flexibility in where computations are made, and decisions taken.
◦ Calvin separates between the different stages of application life-cycle which enables each part to focus on their own specialities.
◦ Requirement-based deployment of applications allows for a greater level of automatic control [**17**].

5

Scheduling of Dataflow Models

5.1 Scheduling Policies

There are several simple and advanced scheduling policies for actors that are using a single core such as Non-Preemptive, Round Robin and other policies. Non-Preemptive and Round Robin are well-known and they are implemented in many systems of different types [**15**].

5.1.1 Non-Preemptive

Non-Preemptive (also known as cooperative scheduling) is a scheduling algorithm that keeps firing the same actor as long as it has available space in the outgoing buffers and the required tokens in the in-ports. When these conditions are no longer fulfilled the next actor in the queue will be fired [**15**]. (See Listing 4.1).

```python
def _fire_actor_non_preemptive(self, actor):
        """
        Try to fire actions on actor on this runtime.
        Returns boolean that is True if actor fired
        """
        # First make sure we are allowed to run
        if not actor._authorized():
            return False
        # Repeatedly go over the action priority list
        done = False
        actor_did_fire = False
        while not done:
            did_fire, output_ok, exhausted = actor.fire()
            actor_did_fire |= did_fire
            if not did_fire:
                # We reached the end of the action list without ANY
    firing during this round
                # => handle exhaustion and return
                actor._handle_exhaustion(exhausted, output_ok)
                done = True
        return actor_did_fire
```

Listing 5.1 The Non preemptive scheduling policy

5.1.2 Round Robin

Round Robin is another scheduling algorithm that spends the same amount of time at each action. After it is done with one action it starts with another one and repeats this process until there are no actions left. By shortening the time spent on the actions the performance of Round Robin can be improved. However, the time spent on each action cannot be too short because then the system will not be able to handle the fast switching of actions [**30**]. In this thesis there is however no need to concern about the length of time since a new actor will not be executed until the first actor is finished, no matter how long time it takes because it is not allowed to interrupt an execution of an action. After an action has been executed all the other actors in the actor list that are going to be fired must be fired once before the first actor can be fired again [**15**]. (See Listing 4.2).

```
def _fire_actor_RoundRobin(self, actor):
        """
        Try to fire action on actor on this runtime.
        Returns boolean that is True if actor fired
        """
        # First make sure we are allowed to run
        if not actor._authorized():
            return False
        did_fire, output_ok, exhausted = actor.fire()
        if not did_fire:
            # => handle exhaustion and return
            actor._handle_exhaustion(exhausted, output_ok)
        return did_fire
```

Listing 5.2 The Round Robin scheduling policy

See the code in Listing 6.1 which calls the methods *fire_actor_round_robin* and *fire_actor_Non_Preemptive*.

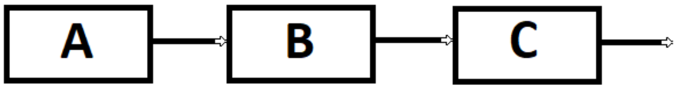

Figure 5.1 ABC Scenario

Figure 6.1 is a simple scenario used to illustrate how the code for Non-preemptive and Round Robin scheduling policies work.

All actors (for instance A, B and C in Figure 6.1) that are ready to fire will be collected in the class NewScheduler (that will be further explained in Chapter 6). NewScheduler class consists of different methods and the method that collects all the actors is called *Strategy*. From *Strategy* the actors will be sent forward one by one to another method, *Non – Preemptive*. The first step in the Non-Preemptive scheduling policy is to initiate *done* and *actor_did_fire* to *false* (See Listing 5.1). In line 12 in Listning 5.1 *while loop* will run until *done* is *true*. In the *while loop* the actors will be fired via actor.fire() method. *Did_fire* is a local variable which will store the boolean value that it will receive from actor.fire(). If the actor did fire then *did_fire* is *true* and the *while loop* will keep running until *did_fire* is *false*. *Done* will be *true* which means that the end of the actions list has been reached for the actor and the *while loop* will be stopped. The method will return either to *true* or *false* (See Listing 5.1). In the scenario in Figure 6.1 the first actor is actor A and when it reaches *while loop* it will stay there until all actions are processed. will be stuck in the *while loop* until all actions are processed. This will happen for the actors B and C too.

The code for Round Robin scheduler policy is even more simple than for Non-Preemptive. The actors will be sent one by one to a method, called Round Robin (See Listing 5.2). The Round Robin method will only try to fire the actor once before it will move on and try to fire the next actor. Whether or not the actor did fire successfully the Round Robin method will return a value as soon as it has tried to fire the actor (*true* means it fired and *false* means it did not fire).

5.2 Scheduler

In the future there will be a limitation in ability to produce faster processes since the processes are decreasing in size. This has caused an increased awareness of dataflow programming in recent years. The concurrency makes it more difficult to create an efficient design for the applications on these new processes. Dataflow programming is useful since it solves the efficiency problem [**31**].

Instructing a network of actors calls *dataflow program*, each actor represent an isolated computational kernel. The actors are connected with each other by buffers. In Calvin the buffers have a limited capacity to store tokens and it has been pre-decided that each buffer will only contain maximum 4 tokens, scheduling policy and the buffer size need to be computed because they affect the efficiency. [**31**].

5.3 The Current Scheduler

There are three different situations that can prevent an actor to produce tokens continuously. The first situation is that every actor has FIFO in- and out-ports. The actors need indata to produce tokens but not all indata come in at the same time, so before firing the actor the scheduler must wait until all indata arrives. The second situation is when an actor waits for an event to occur, for example when a button is pressed and sends the data to the actor. The third situation is when a timer is used to send data repeatedly at a certain time. The scheduler is very important for the application execution, for every runtime it is responsible for data transport and triggering actor actions on indata. The current scheduler is a very simple Non-Preemptive (NP) scheduler. It knows what data has arrived from other runtimes, the rules for triggering an action, and the output data to be passed on to other actors whether they are local or remote. The scheduler consists of a BaseScheduler class, which includes a subclass that is called SimpleScheduler. SimpleScheduler uses BaseScheduler and the BaseScheduler contains the following important methods:

- Run(self): The system starts.

- Stop (self): The system exits.
- strategy(self): The scheduling part will be in this method.
- watchdog(self): If there is nothing to schedule then the watchdog will be called.
- insert_task(self, what, delay): This method will insert tasks in time order.
- _schedule_next(self, delay, what): This method will schedule the next task in the queue.
- _process_next(self): Here the tasks will be processed.
- _fire_actors(self, actors): The method will try to fire actions on a list of actors.
- _fire_actor(self, actors): The method will try to fire actions on an actor. it will be called from the _fire_actors method.

The new strategies will be based on SimpleScheduler class. It contains the following methods:

- tunnel_rx(self, endpoint): This method calls self.insert_task when a token is received on an endpoint.
- tunnel_tx_ack(self, endpoint): When it has successfully received an ACK on the sent token it will call the methods self.monitor.clear_backoff and self.insert_task.
- tunnel_tx_nack(self, endpoint): When it received an NACK on sent token it will call the methods self.monitor.set_backoff and self.insert_task.
- schedule_calvinsys(self, actor_id=None): If it receives a token from a sensor then it will call the self.insert_task.
- strategy(self): It will try to fire all actors all the time.
- watchdog(self): It will send a log and calls self.insert_task.

The SimpleScheduler is a very simple scheduler which will try every enabled actor in the runtime, and it works as following:
1) Get a list over all enabled actors e.g. if we have an application which consist of two actors so these actors will be enabled.
2) SimpleScheduler uses Non-Preemptive policy which as mentioned before will try to fire every actor to produce tokens.
3) The scheduler will try to send all produced tokens from all actors.
If there is nothing else scheduled, the watchdog will be activated.

6

Implementation of the New Scheduler

6.1 New scheduling strategy

The main idea of the new strategy is that instead of trying to fire all actors every time, the scheduler will be called to make sure that the most appropriate actor/actors will be fired. The scheduler will sort the available actors dynamically which means that it will pick the actors that are necessary to fire. The scheduler chooses the actors that have received all the necessary tokens.

For instance assume that there are three actors A, B and C, where A is connected to B and B is connected to both A and C. The first time the scheduler will try to fire all actors (A, B and C). The next time the scheduler will find and fire actors that are connected to the actors that were just fired. This means that in this case the scheduler will try to find actors that are connected to the actors A, B and C. If only actor A were fired the previous time, then the scheduler will put actor B to the list of actors that will fire next, since actor B is the only actor that is connected to actor A. After actor B has been fired the scheduler will put the actors that are connected to actor B (that is A and C) to the list of actors that will be fired next and so on see Figure 6.1.

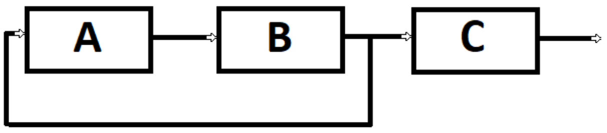

Figure 6.1 The idea of how the new strategy works.

NewScheduler is a class that consists of several methods (see Listing 6.1). The most important methods are *schedule_calvinsys* and *strategy*. *Schedule_calvinsys* includes actors that will be processed as soon as they have reached the *next_round* list. For instance, if the actor is a button then it is important to process the button as soon as it is pressed. The *strategy* method will be called with no delay via *insert_task* method. If the *this_round* list in the *strategy* method is empty, then all actors will be added in *this_round* list. The produced tokens will be sent via *monitor.communicate* to the out ports. *Did_fire_actors* is a list that collects only the actors that have been fired of all the actors in *this_round* list. *Next_round* list collects the actors that are connected to the actors in *did_fire_actors* via out-port. The *strategy* method will be called again via *insert_task* method and so on.

```
class NewScheduler(SimpleScheduler):
    def __init__(self, node, actor_mgr):
        super(NewRoundRobinScheduler, self).__init__(node, actor_mgr)
        self.next_round = set()
        self.this_round = set()
        self.endpoints = []

    def schedule_calvinsys(self, actor_id=None):
        """Incoming platform event"""
        if actor_id:
            try:
                actor = self.actor_mgr.actors[actor_id]
                if actor.enabled():
                    self.next_round.add(actor)
            except:
                pass
        self.insert_task(self.strategy, 0)

    def strategy(self):
        self.this_round = self.next_round
        self.next_round = set()

        # New_Strategy
        if self.this_round == set():
            # all actors to fire
             self.this_round = self.actor_mgr.enabled_actors()
             self.endpoints = self.monitor.endpoints
        # Communicate
        did_transfer_tokens = self.monitor.communicate(self.endpoints
    )

        # ids from actors did fire
        did_fire_actors = [actor for actor in self.this_round if self
    ._fire_actor_non_preemptive(actor)]
        _log.info("Did fire actors:\n%s\n%s" % (str([actor._name for
    actor in self.this_round]), str([actor._name for actor in
    did_fire_actors])))
```

```
        # get next actors we want to fire next
        self.next_round, self.endpoints = self.get_next_actors(did_fire_actors)
        # Repeat if there was any activity
        activity = did_transfer_tokens or bool(did_fire_actors)
        n = time.time()
        _log.info("activity %s %s" % (activity, str([(t[0] - n, t[1].__name__ if hasattr(t[1], "__name__") else "other") for t in self._tasks])))
        if activity:
            self.insert_task(self.strategy, 0)

    def get_next_actors(self, actors):
        next_actors = set()
        endpoints = []
        for actor in actors:
            for outport in actor.outports.values():
                endpoints.extend(outport.endpoints)
                for oep in outport.endpoints:
                    try:
                        next_actors.add(oep.peer_port.owner)
                    except:
                        pass
        return next_actors, endpoints
```

Listing 6.1 The NewScheduler

6.2 Results

The total time for producing 5000 tokens for the different scenarios are used to analyze the behavior of the current scheduling policy, Non-Preemptive. The results have been compared to the results of the Round Robin Scheduling policy and to the new strategies using Non-Preemptive and Round Robin with dynamical sorting. The results are plotted for when 1, 2, 5, 10, 20, 50 and 100 applications run at the same time.

Figure 6.2 shows the result of using Source of type std.Trigger with delay of 0.5 seconds and nbr = 1. The new strategies using dynamical sorting for both Round Robin Scheduling policy and Non-Preemptive Scheduling policy take much less time to produce 5000 tokens than Non-Preemptive and Round Robin without dynamical sorting, independently on the number of application.

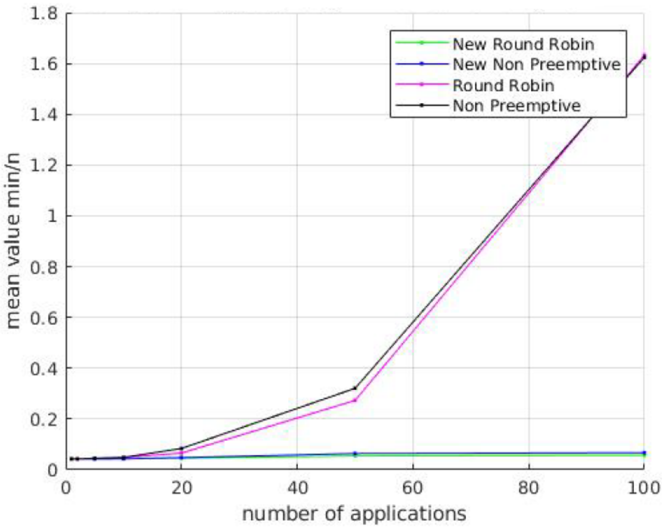

Figure 6.2 Using std.Trigger(0.5), n=5000 Tokens and nbr = 1

In Figure 6.3 nbr is changed to 4. As seen in the figure the time increases if more applications are running. The mean time of Round Robin and Non Preemptive increase much more than the mean time of the new strategies when increasing the number of the applications that run at the same time which can be seen in Figure 6.2 and Figure 6.3.

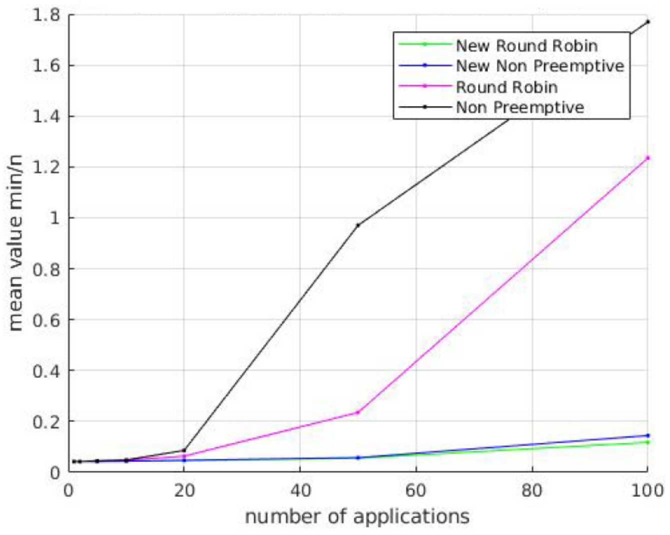

Figure 6.3 Using std.Trigger(0.5), n=5000 Tokens and nbr = 4

Figure 6.4 shows the results of using the Source Actor of type std.Counter and nbr = 1.
The time for all four strategies increase as the number of running applications increases, but they do not increase as fast as when using std.Trigger. For instance if we compare the Non-Preemptive at 100 applications we see that the highest mean time is approximately 1.8 minutes when using std.Trigger and 0.12 minutes when using std.Counter.

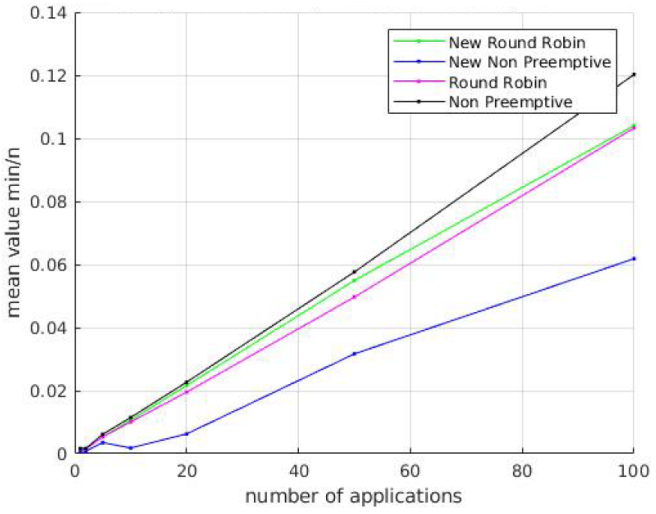

Figure 6.4 Using std.Counter, n=5000 Tokens and nbr = 1

In the last Figure 6.5 the mean time for all four strategies increase, but for Non Preemptive with dynamical sorting the mean time begins to decrease after 50 applications instead of increasing. This was unexpected and can be caused by a bug or that something went wrong during the collection of data.

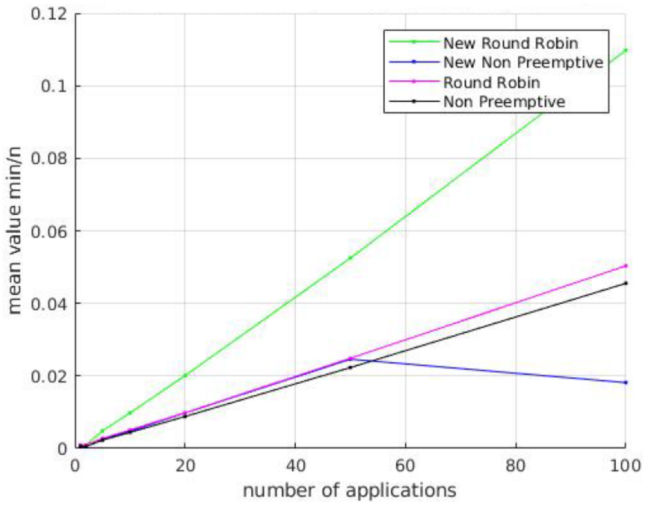

Figure 6.5 Using std.Counter, n=5000 Tokens and nbr = 1

Another way to analyze the behaviour of the different strategies is by analyzing the Profile results. What we get of the profile results is the number of the calls for a specific function, the total time the function takes to do a specific job (for this thesis a job is letting 5000 tokens pass through all actors for 10 application runs in the same time). The profile also measures the cumulative time spent in a specific function. Appendix 6.6 shows the results of three functions: Strategy _fire_actor_once or _fire_actor_non_preemptive and communicate. The three functions are approximately what the strategy part consists of in the scheduler. The results are analyzed by comparing the different ncalls, tottime and cumtime for every function compared to the Non-Preemptive scheduling policy. Note the Different scale for Figures.6.2 and 6.5. Figures 6.6 to 6.10 shows the graphs of the results of profiling when the source actor in the scenario is of type std.Trigger. Figures 6.11 to 6.15 shows the graphs of the results of profiling when the source actor in the scenario is of type std.Counter.

When using Source Actor type std.Trigger the total time strategy almost remained the same for nbr = 1 and nbr = 4 for Round Robin and for New Non-Preemptive. The total time strategy is slightly higher for nbr = 1 (1,394) compared to nbr = 4 (1,383) for New Round Robin. There is a significant difference between the total time strategy for nbr = 1 (1,397) and nbr = 4 (1,297) for Non-Preemptive (See Figure 6.6).

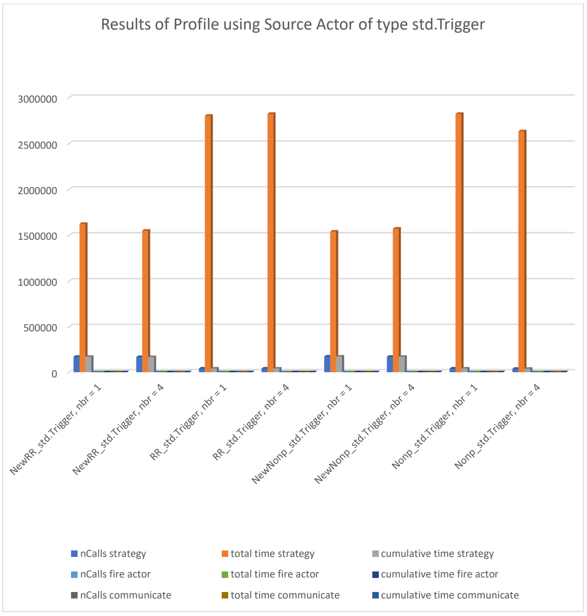

Figure 6.6 Results from the profiling, using std.Trigger as a source

When using Source Actor type std.Trigger the nCalls strategy and the cumulative time strategy are almost the same for nbr = 1 and nbr = 4 for Round Robin. The nCalls strategy and the cumulative time strategy are slightly higher for nbr = 1 compared to nbr = 4 for New Round Robin (170636 and 202,213 for nbr = 1 versus 167324 and 195,478 for nbr = 4), New Non-Preemptive (173258 and 207,220 for nbr = 1 versus 170242 and 211,436 for nbr = 4) and Non-Preemptive (40690 and 317,619 for nbr = 1 versus 52130 and 102,793 for nbr = 4) (See Figure 6.7).

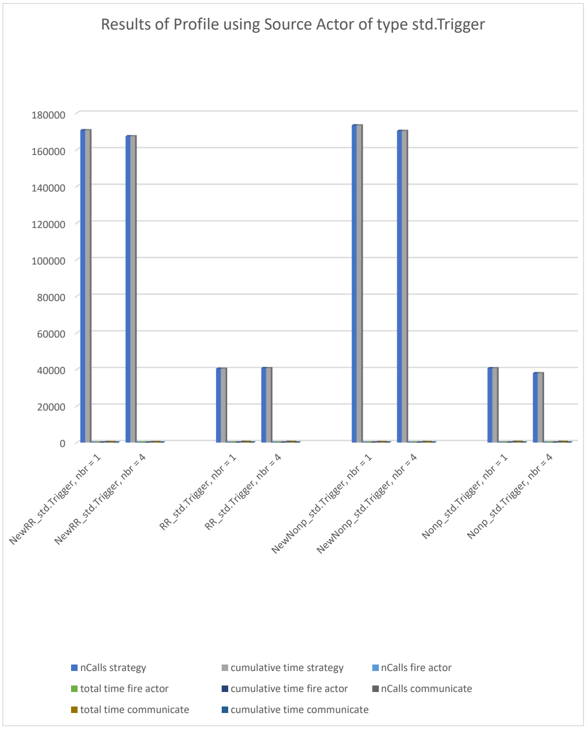

Figure 6.7 Results from the profiling, without total time strategy

When using Source Actor type std.Trigger the nCalls communicate and the total time communicate are almost the same for nbr = 1 and nbr = 4 for all four scheduling strategies. The nCalls communicate and the total time communicate are lower for New Round Robin compared to the other three scheduling strategies (See Figure 6.8).

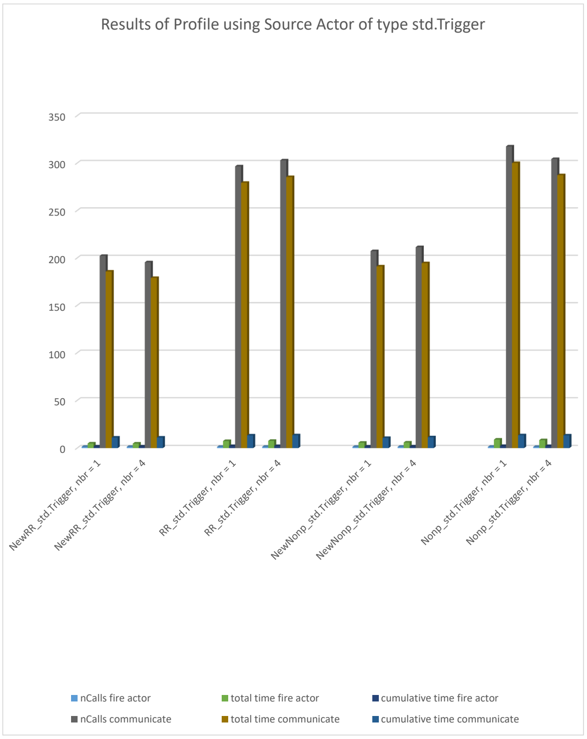

Figure 6.8 Results from the profiling, without nCalls strategy and cumulative time strategy

When using Source Actor type std.Trigger the total time fire actor and the cumulative time communicate are lower for New Round Robin and New Non-Preemptive compared to Round Robin and Non-Preemptive. (See Figure 6.9 and Appendix).

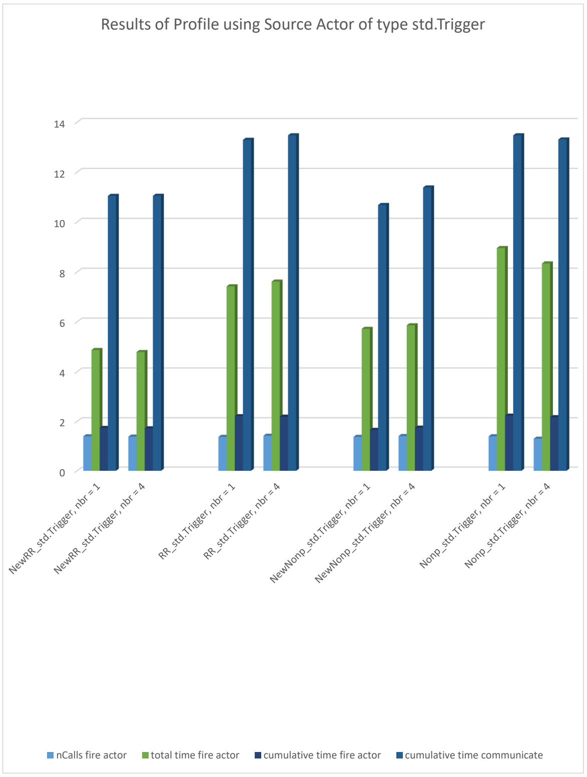

Figure 6.9 Results from the profiling, without nCalls and total time communicate

When using Source Actor type std.Trigger the nCalls fire actor is the same for all four scheduling strategies for both nbr = 1 and nbr = 4. The cumulative time communicate is lower for New Round Robin and New Non-Preemptive compared to Round Robin and Non-Preemptive (See Figure 6.10 and Appendix).

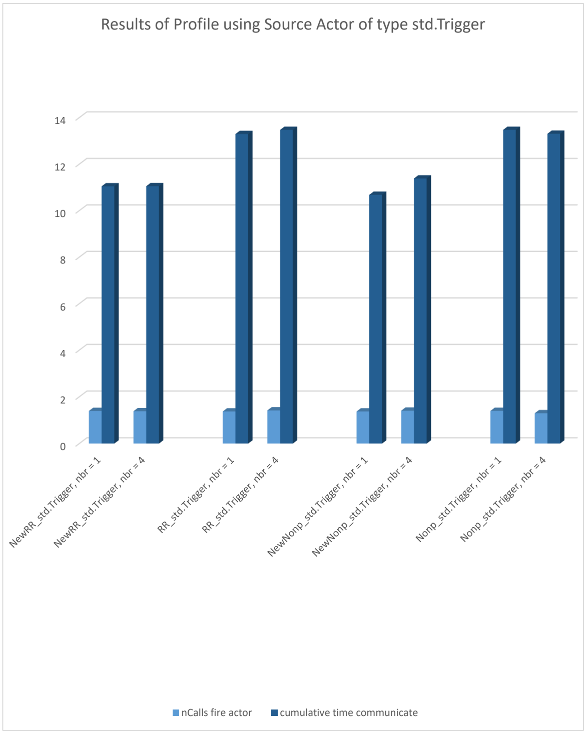

Figure 6.10 Results from the profiling, without total time fire actor and cumulative time fire actor

When using Source Actor type std.Counter the nCalls fire actor almost remained the same for nbr = 1 and nbr = 4 for new Round Robin. The nCalls fire actor is much higher for nbr = 1 (732578) compared to nbr = 4 (379792) for Round Robin. There is a significant difference between the nCalls fire actor for nbr = 1 (266067) and nbr = 4 (109867) for new Non-Preemptive, and we can see that it has the lowest value of the whole graph for both nbr = 1 and 4. The nCalls fire actor is much higher for nbr = 1 (725333) compared to nbr = 4 (192717) for Non-Preemptive (See Figure 6.11).

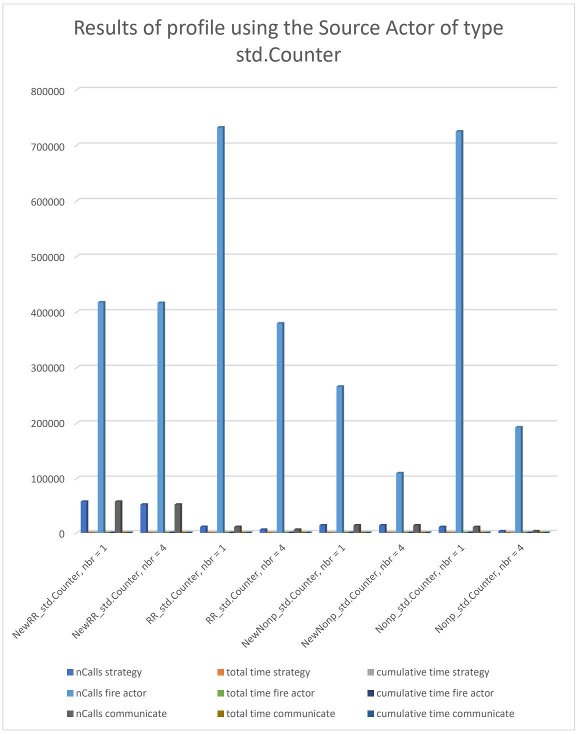

Figure 6.11 Results from the profiling, using std.Counter as a source

When using Source Actor type std.Counter the nCalls strategy and nCalls communicate are lowest for Non-Preemptive nbr = 4 (See Figure 6.12).

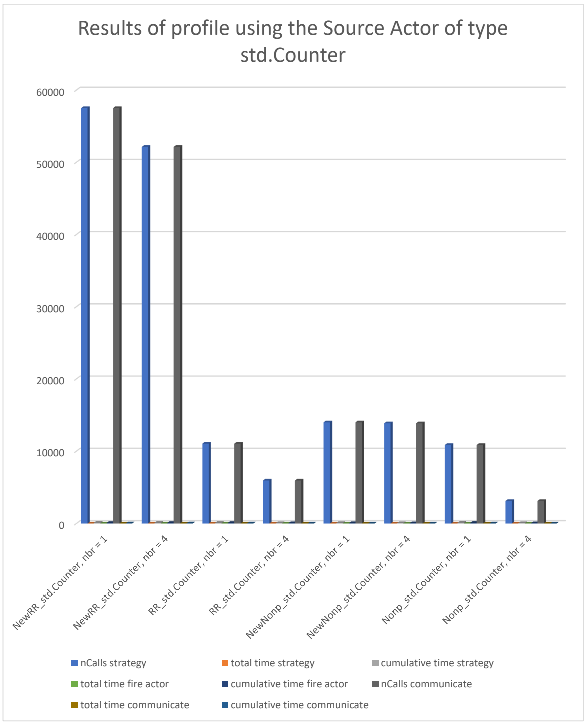

Figure 6.12 Results from the profiling, without nCalls fire actor

When using Source Actor type std.Counter new Round Robin have higher cumulative time strategy, cumulative time fire actor and cumulative time communicate for both nbr = 1 and nbr = 4 compared to Round Robin. The cumulative time strategy and the cumulative time fire actor are higher for Non-Preemptive nbr = 1 compared to New Non-Preemptive nbr = 1, but almost the same for Non-Preemptive nbr = 4 compared to New Non-Preemptive nbr = 4. The cumulative time communicate is the lowest for Non-Preemptive nbr = 4 comparing to all the other scheduling strategies (See Figure 6.13).

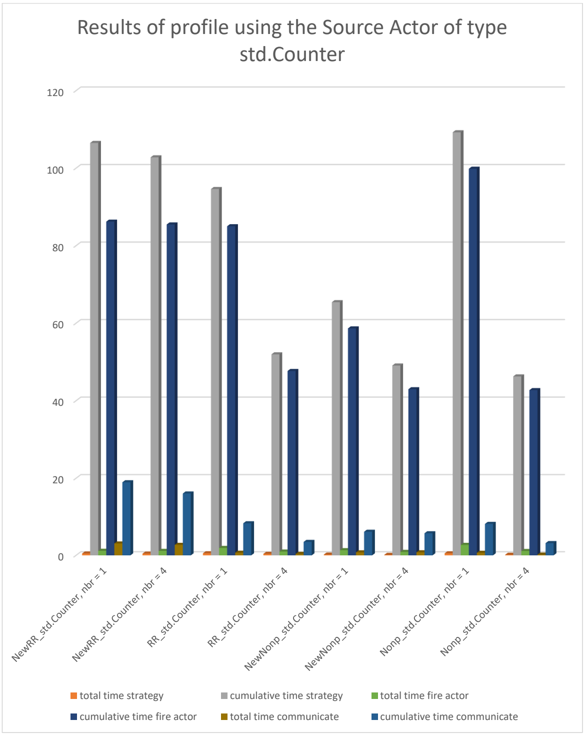

Figure 6.13 Results from the profiling, without nCalls strategy and nCalls communicate

When using Source Actor type std.Counter new Round Robin have higher cumulative time communicate and total time communicate for both nbr = 1 and nbr = 4 compared to Round Robin. The cumulative time communicate is slightly higher for Non-Preemptive with nbr = 1 and slightly lower for Non-Preemptive with nbr = 4 compared to New Non-Preemptive with both nbr = 1 and nbr = 4. The total time communicate is almost the same for New Non-Preemptive and Non-Preemptive (See Figure 6.14).

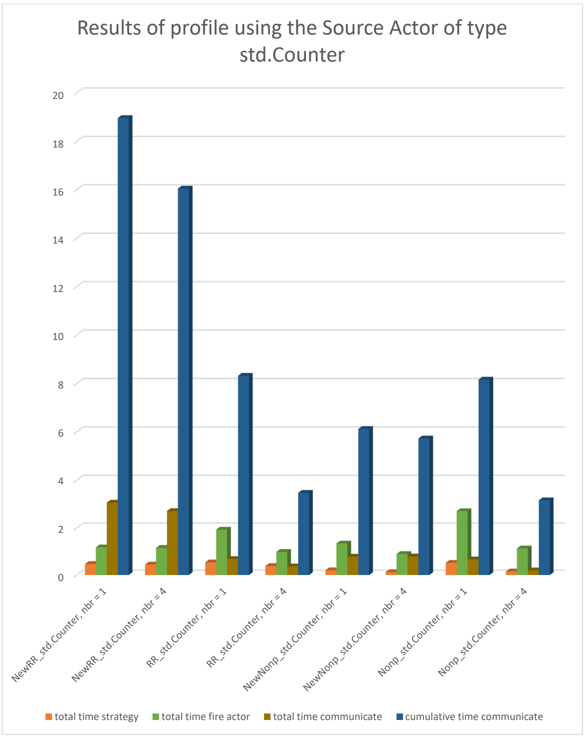

Figure 6.14 Results from the profiling, without cumulative time strategy and cumulative time fire actor

When using Source Actor type std.Counter Non-Preemptive have the highest total time fire actor compared to all the other scheduling strategies. Round Robin with nbr = 1 have higher total time fire actor than New Round Robin. The total time strategy is pretty much the same for both New Round Robin (nbr = 1 and nbr = 4) and Round Robin (nbr = 1 and nbr = 4). Non-Preemptive with nbr = 1 has higher total time strategy compared to New Non-Preemptive (See Figure 6.15).

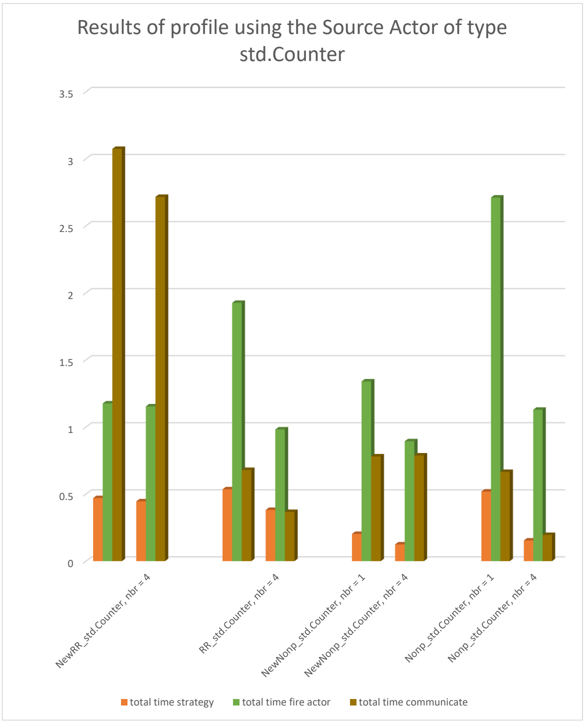

Figure 6.15 Results from the profiling, without cumulative time communicate

6.3 Discussion

The new strategy using Round Robin Scheduling policy and the new strategy using Non-Preemptive Scheduling policy take much less time to produce 5000 tokens independently on the application quantity when using source of type std.Trigger with nbr = 1 and 4 than the other two Round Robin and Non-Preemptive scheduling policies. This is because the New strategy choses the right actors that will be fired as described in Section 5.2 instead of trying to fire all available actors. As seen in Figure 6.3 when changing nbr to 4 it means that there are four tokens that will arrive to the actor inport at the same time instead of just one token at a time. That means that the in-port queue for an actor will be full and the system will not be able to send four new tokens until the previous four tokens have been processed and the queue is empty again. It seems like the new strategies are much better than Non-Preemptive and Round Robin when it is some time delay between the sent tokens.

In the other situation when using a source actor of type std.Counter there is no time delay between the received tokens. It means that 5000 tokens are trying to be pushed through the whole application as fast as possible. In the plot for nbr = 1 in Figure 6.4 it can be seen that the New Non-Preemptive has less steep slope than the other three slopes. For nbr = 4 the new Round Robin has a greater slope than the other three which means that it takes longer time. The New Non-Preemptive does not behave as expected because its slope decreases as the quantity of running applications increases. The reason for this unexpected behaviour is unknown and when the collection of data and plotting was repeated the result remained the same. Therefore, profiling is used to ensure that the results are reliable.

In general it can be seen from Figure 6.6 to Figure 6.10 that the total time for strategy, fire actor and communicate functions for New Round Robin and New Non-Preemptive are lower than for Round Robin and Non-Preemptive. This means that the new strategies are better because they produce 5000 tokens in less time when we use std.Trigger as a source since the system has not much load on it which means the actors are not busy all the time. In Figure 6.7 the nCalls strategy and the cumulative time strategy are much higher in the new strategies than in Round Robin and Non-Preemptive and that is because Round Robin and Non-Preemptive try to fire every available actor, while the new strategies try to fire only a few actors that have the highest probability to be fired. It takes more calls to strategy to find the actors that actually can be fired. It can be seen that Figure 6.6 to Figure 6.10 consist with Figures 6.2 and 6.3.

The other scenario when using std.Counter as a source has a property that the actors are always busy. The total time for strategy, fire actor and communicate functions are compared with each other in Figures 6.11 to 6.15. For all strategies it can be seen that the sum of all the Non-Preemptive total times for nbr = 4 is 1,639 which is lower than the sum of all the total times for New Non-Preemptive which is 1.8 see Figure 6.15. From Figure 6.9 it can also be seen that Non-Preemptive has the lowest cumulative total time and from Figure 6.13 it can be seen that it has the lowest cumulative time strategy as well. Figure 6.12 shows that Non-Preemptive also have the lowest nCalls strategy and nCalls communicate. This means that the Non-Preemptive is best when the actors have much to do.

6.4 Conclusion

New Non-Preemptive and New Round Robin are beneficial to use when the most actors are not busy for every strategy function call. This applies when std.Trigger is used as a source because there is a time delay between the tokens and the system will not need to wait for the actors to process the tokens because the actors will have enough time to process them. However, when std. Counter is used as a source it is more beneficial to use Non-Preemptive scheduling policy as a strategy because in this situation the actors will be busy all the time and skipping search after the best actors to fire will save more time to process the tokens.

6.5 Future Work

There are a lot of possibilities for future work in the different areas of Calvin. For Calvin's scheduler these results concluded that the new scheduler is not always beneficial, but only in some scenarios. Future work could focus in building a method that easily can change between the old and the new scheduler depending on the load, since it is a better to use the old scheduler that will process all tokens if there is a high load on actors and better to use the new scheduler that will select tokens to process if there is a low load on actors. The method could be inspired by for instance NIC (Network Interface Controller) that uses a similar switch between poll and event-based modes depending on high or low loads.

Bibliography

[1] *Calvin,https://www.ericsson.com/research-blog/open-source-calvin/.* viewed 2017-11-04

[2] *Xia, F. Yang, L. Wang, L. Vinel, A.,Internet of Things,Int. J. Commun. Syst. 2012; 25:1101–1102 DOI:10.1002/dac URL:https:// pdfs.semanticscholar.org/930c/4981e87584afa7e6f1f4977323e365 aae097.pdf*

[3] *weber, R.H. weber, R. (2010). "Internet of things", Springer Heidelberg Dordrecht London New York, pp. 1–2.ISBN 978-3-642-11709-1 .DOI 10.1007/978-3-642-11710-7 .*

[4] *Amardeep, M.; Baddour, R.; Svensson, F.; Gustafsson, H.; Elmorth, E. Calvin Constrained – A Framework for IoT Applications in Heterogeneous Environments, 2017, 1063-6927. DOI 10.1109/ICDCS.2017.181*

[5] *https://www.stackoverflowbusiness.com/blog/the-difference-between-programming-frameworks-and-languages/.* viewed 2018-01-20

[6] *Hewitt, C., P. Bishop, and R. Steiger (1973)". In: Proceedings of the 3rd International Joint Con- ference on Artificial Intelligence. IJCAI'73. Morgan Kaufmann Publishers Inc.,Stanford, USA, pp. 235–245.URL:http://dl.acm.org/citation.cfm?id=1624775.1624804*

[7] *Karmani, R. K., A. Shali, and G. Agha (2009). "Actor frameworks for the JVM plat- Proceedings of the 7th International Confer-ence on Principles and Practice of Programming in Java. PPPJ '09. ACM, Cal-gary, Alberta, Canada, pp. 11–20.ISBN:978-1-60558-598-7.DOI:10.1145/1596655.1596658.*

[8] *Eker, J. and J. W. Janneck (2003). rep. UCB/ERL M03/48. EECS Department, University of California, Berkeley.URL:http://www2.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html*

[9] *Charousset, D., T. C. Schmidt, R. Hiesgen, and M. Wählisch (2013). In:Proceedings of the 2013 Workshop on Programming Based on Actors, Agents,and Decentralized Control. AGERE! 2013. ACM, Indianapo-*

lis, Indiana, USA,pp. 87–96. ISBN: 978-1-4503-2602-5. DOI: 10. 1145 / 2541329.2541336.URL: http://doi.acm.org/10.1145/2541329.2541336.

[10] *STEWART, J. 2014. Python for Scientists.Cambridge: Cambridge University Press, p.2*

[11] *Python,https://www.python.org/doc/essays/blurb/.* viewed 2017-11-04

[12] *Profilers ,https://docs.python.org/2/library/profile.html.* viewed 2018-01-21

[13] *Sale, D., (2014), Testing Python ,John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom ,pp. 1-7.*

[14] *Persson, P. Angelsmark, O. Calvin–Merging Cloud and IoT, Procedia Computer Science, Volume 52, pp. 210-217,2015. URL:https://www.sciencedirect.com/science/article/pii/S1877050916308948*

[15] *Michalska M. , Zufferey N., Boutellier J., Bezati E.,Mattavelli M. Efficient scheduling policies for dynamic dataflow programs executed on multi-core, Volume 52, pp. 210-217. URL:http://research.ac.upc.edu/multiprog/multiprog2016/papers/multiprog-2.pdf*

[16] *Deployment,https://www.ericsson.com/research-blog/closer-look-calvin/.* viewed 2018-01-15

[17] *Persson, P. Angelsmark, O. Requirement-Based Deployment of Applications in Calvin, pp. 79-81,2017. URL:https://link-springer-com.ludwig.lub.lu.se/content/pdf/10.1007%2F*$978-3-319-56877-5\_5$*.pdf*

[18] *Actors,https://github.com/EricssonResearch/calvin-base/wiki/Actors.* viewed 2018-01-11

[19] *Actors,https://github.com/EricssonResearch/calvin-base/wiki/Actor-Docs.* viewed 2017-12-11

[20] *capecode,https://chess.eecs.berkeley.edu/capecode/.* viewed 2018-01-20

[21] *Accessors,https://www.icyphy.org/accessors/.* viewed 2018-01-11

[22] *Ptolemy II,http://ptolemy.eecs.berkeley.edu/ptolemyII/.* viewed 2018-02-04

[23] *accessosrs,https://www.icyphy.org/accessors/wiki/Main/CapeCodeHost#toc1.* viewed 2018-01-21

[24] *AWS Lambda, https://aws.amazon.com/lambda/features/.* viewed 2018-02-07

[25] *https://s3.amazonaws.com/awslambda-reference-architectures/iot-backend/ lambda-refarch-iotbackend.pdf.* viewed 2018-02-07

[26] *IBM Bluemix−Next−Generation Cloud App Development Platform, 04 2017, Available: https://console.ng.bluemix.net/.*viewed 2018-02-3

[27] *https://nodered.org/#features.*viewed 2018-02-3

[28] *Cory, Gackenheimer. 2013. Node.js Recipes. A Problem Solution Approach. 1uppl http://pdf.th7.cn/down/files/1407/Node.js%20Recipes.pdf.*viewed 2018-02-3

[29] *Silberschatz, Gagne, Galvun, (2009), Operating System Concepts, 8th Edition,John Wiley and Sons Inc. USA,pp. 185-186.*

[30] *Round Robin,http://pages.cs.wisc.edu/ remzi/OSTEP/cpu-sched.pdf, p.7-8.* viewed 2018-01-11

6.6 Appendix

Profile Results

```
-------------------------------------------------------------------------------
NewRR_Trigger = 0.05
nbr = 4:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall       function
167324   1.383    195.478    0.001         strategy
1551307  4.786    178.966    0.000         _fire_actor_once
167324   1.717     11.050    0.000         communicate
-------------------------------------------------------------------------------
NewRR_Trigger = 0.05
nbr = 1:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall       function
170636   1.394    202.213    0.001         strategy
1625110  4.868    185.691    0.000         _fire_actor_once
170636   1.735     11.046    0.000         communicate
-------------------------------------------------------------------------------
RR_Trigger = 0.05
nbr = 4:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall       function
40775    1.418    302.940    0.007         strategy
2824234  7.609    285.328    0.000         _fire_actor_once
40775    2.190     13.473    0.000         communicate
-------------------------------------------------------------------------------
RR_Trigger = 0.05
nbr = 1:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall       function
40439    1.373    296.611    0.007         strategy
2803430  7.420    279.272    0.000         _fire_actor_once
40439    2.209     13.294    0.000         communicate
-------------------------------------------------------------------------------
NewNonp_Trigger = 0.05
nbr = 4:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall        function
170242   1.406    211.436    0.001          strategy
1573371  5.859    194.533    0.000          _fire_actor_non_preemptive
170242   1.745     11.383    0.000          communicate
-------------------------------------------------------------------------------
NewNonp_Trigger = 0.05
nbr = 1:
-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall        function
173258   1.373    207.220    0.001          strategy
1541336  5.715    191.105    0.000          _fire_actor_non_preemptive
173258   1.655     10.681    0.000          communicate
-------------------------------------------------------------------------------
```

```
Nonp_Trigger = 0.05
nbr = 4:
-------------------------------------------------------------------------------
ncalls    tottime   cumtime    percall        function
37988     1.297      304.344   0.008          strategy
2634478   8.338      287.263   0.000          _fire_actor_non_preemptive
37988     2.170       13.307   0.000          communicate
-------------------------------------------------------------------------------
Nonp_Trigger = 0.05
nbr = 1:
-------------------------------------------------------------------------------
ncalls    tottime   cumtime    percall        function
40690     1.397      317.619   0.008          strategy
2824045   8.954      300.095   0.000          _fire_actor_non_preemptive
40690     2.232       13.473   0.000          communicate
```

Profile Results

```
-------------------------------------------------------------------------------
NewRR_Counter
nbr = 4:

-------------------------------------------------------------------------------
ncalls   tottime  cumtime    percall       function
52130    0.446    102.793    0.002         strategy
416731   1.154     85.466    0.000         _fire_actor_once
52130    2.716     16.046    0.000         communicate
-------------------------------------------------------------------------------
NewRR_Counter
nbr = 1:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime    percall       function
57508    0.470   106.507    0.002         strategy
417672   1.176    86.196    0.000         _fire_actor_once
57508    3.074    18.961    0.000         communicate
-------------------------------------------------------------------------------
RR_Counter
nbr = 4:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime    percall       function
5938    0.382    51.962     0.009         strategy
379792  0.982    47.665     0.000         _fire_actor_once
5938    0.367     3.481     0.001         communicate
-------------------------------------------------------------------------------
RR_Counter
nbr = 1:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime   percall       function
11035   0.536    94.615    0.009         strategy
732578  1.926    85.017    0.000         _fire_actor_once
11035   0.680     8.315    0.001         communicate
-------------------------------------------------------------------------------
NewNonp_Counter
nbr = 4:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime    percall      function
13856   0.125    49.076     0.004        strategy
109867  0.894    42.961     0.000        _fire_actor_non_preemptive
13856   0.788     5.724     0.000        communicate
-------------------------------------------------------------------------------
NewNonp_Counter
nbr = 1:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime   percall       function
13988   0.203    65.423    0.005         strategy
266067  1.340    58.657    0.000         _fire_actor_non_preemptive
13988   0.781     6.118    0.000         communicate
```

```
-------------------------------------------------------------------------------
Nonp_Counter
nbr = 4:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime   percall       function
3117    0.154    46.268    0.015         strategy
192717  1.129    42.727    0.000         _fire_actor_non_preemptive
3117    0.195     3.172    0.001         communicate
-------------------------------------------------------------------------------
Nonp_Counter
nbr = 1:
-------------------------------------------------------------------------------
ncalls  tottime  cumtime    percall       function
10856   0.519    109.249    0.010         strategy
725333  2.711     99.852    0.000         _fire_actor_non_preemptive
10856   0.665      8.156    0.001         communicate
```

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER'S THESIS *Date of issue* September 2018 *Document Number* TFRT-6065
Author(s) Sam Jabbar	*Supervisor* Per Persson, Ericsson Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden Anton Cervin, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Scheduling Strategies for the Calvin IoT Environment

Abstract

The connection of devices to the internet is referred to as Internet of Things (IoT). By using IoT distributed network of devices will be able to communicate with each other. The management and forming of distributed applications is, however, very complex since IoT uses devices that have different capabilities and that do not have the same communication protocols. Calvin is an open source application environment developed by Ericsson that provides a distributed cloud for IoT. The distributed cloud can help to achieve low latency by using parallel processing nodes. Calvin is built upon the actor model, using the methodology dataflow programming. Using Calvin's programming model, the different parts of the system are represented by actors that are used to isolate and abstract functionality. Actors communicate with each other by message passing, however there is a network of runtimes that handle the actual dataflow. Several runtimes are applied in every device that will be used in IoT to get metadata about the capabilities of the devices and to get more generic and reusable actors [1].

In Calvin the scheduler is responsible for data transport and for triggering actor actions on in-data. The scheduler is necessary for execution of applications for each runtime. The current scheduler is a basic Non-Preemptive (NP) scheduler with limited knowledge because it only uses local information, such as what data has arrived from other runtimes, the rules for triggering an action, and the output data that will be sent forward to other actors whether they are local or remote. The aim of the master thesis is to improve Calvin's current scheduler. A new scheduling policy called Round Robin will be implemented and compared with the current Nonpreemptive scheduling policy with respect to latency. A dynamic sorting part will be added to Calvin's scheduler and implemented for the two scheduling policies. The result of this thesis shows that the new strategies named New Non-Preemptive and New Round Robin are beneficial to use when most actors are not busy for every function call.

Keywords

Calvin, Internet of Things, Dataflow programming, distributed application,
Actor model, scheduling, Round Robin, Non-preemptive.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-62	*Recipient's notes*
Security classification		