Heron

George Dimitriadis

May 03, 2024

ABOUT

1 About the Heron framework **1**

2 Installation & Startup **3**
2.1 Installation 3
2.2 Startup 6

3 The Editor **7**
3.1 Node Menu and Node Editor Window 7
3.2 Menu Bar 8
3.3 Secondary Node Window 8
3.4 The Graph Control Buttons and the Nodes' life cycle 9

4 Setting up and using multiple machines **11**
4.1 Setting up the machines 11
4.2 Setting up the LAN in Heron 12
4.3 Setting up the pipeline 12
4.4 Multiple environments 14

5 All about the Nodes **15**
5.1 The scripts 17
5.2 The Node types 17
5.3 Creating a Graph (Pipeline) 17
5.4 Running a Graph (a Node's life) 17
5.5 The Heartbeat System 18
5.6 The inner workings 19
5.7 Choosing the CPU core 21

6 Adding Repositories as new Nodes **23**
6.1 Adding an existing repository 23
6.2 Downloading a repository from inside Heron 23
6.3 Creating a valid Heron Nodes repository from scratch 23

7 Writing new Nodes **27**
7.1 The Node templates 27
7.2 The com script 27
7.3 The worker script 29

8 In Node Visualisation **35**
8.1 The API 35
8.2 The visualisation elements 36

9 The Saving State System **41**
9.1 A Node's Saving System 41
9.2 Implementation in the Node 41

10 Synchronisation **45**
10.1 Hardware 45
10.2 Software 45
10.3 Examples 46

11 Debugging **49**
11.1 Usage and limitations of the python debugging tools 49
11.2 The logging system 49
11.3 Hanging processes 50

12 Indices and tables **51**

CHAPTER

ONE

ABOUT THE HERON FRAMEWORK

Heron is an open source software conceived as an experiment (and general data) pipeline implementation tool. It allows users to realise their conceptual experimental designs such that the final result, i.e. the implementation running the experiment, visually and structurally bears a significant resemblance to its mental schema, i.e. the idea of the experiment in the experimenter's imagination.

Heron combines a Node based experiment designer with a Python API allowing users to quicly construct and add their own Nodes. It fully obfuscates the Node communication from the user and offers seamless communication between separate machines running different operating systems.

CHAPTER

TWO

INSTALLATION & STARTUP

2.1 Installation

2.1.1 The standard way

Heron can be installed as a pip package. Just do:

```
python -m pip install heron-42ad
```

(or whatever python command you use in your system - python3, py, etc.) in any python environment you want. This will install all the required libraries and Heron itself and give you a fully working system.

Note: Heron is not a install and forget type of software. As a user you will have to develop your own Nodes in the form of git repositories and link their code to Heron. So knowledge of where your system has deposited Heron (and more importantly it's Operations folder where the Nodes code resides) will be beneficial.

At some point we also plan to release a conda package, but currently this is lower on the priority list.

2.1.2 The hardcore way (as of May 03, 2024)

Heron was designed so that it doesn't require installation. You can download the code from its github repository and just run it (check out the Startup paragraph on what running Heron actually means).

The same goes for putting the Heron code on machines other than the one running the Heron GUI (in order to use Heron's network functionality). All you have to do is copy the code from the repository to somewhere in your machines that your main machine can have access to through SSH (see info on how Heron connects different machines in the *Setting up and using multiple machines*)

Requirements

In order for the 'just copy the code over' idea to work, you will need to have a python environment that can support Heron's needs. The minimal environment for Heron to start its GUI and also use all its pre packaged Nodes is the following:

1. numpy
2. pandas
3. pyzmq > 20.x
4. paramiko

5. h5py
6. tornado
7. pynput
8. pyserial
9. dearpygui > 1.2
10. opencv >= 4.x
11. psutil

All of the above, except dearpygui and opencv, can be installed through conda (pynput and pyserial are in the conda-forge). So once you set up conda (either miniconda or anaconda) you can do:

```
conda install package_name
```

or

```
conda install -c conda-forge package_name
```

for the packages that can be found the the conda-forge.

You can get dearpygui with a:

```
pip install dearpygui
```

It won't bother the rest of the conda installations. For OpenCV read on.

If the environment you are setting up is for a machine that will not call Heron's GUI then dearpygui is not required. Also check which Nodes you will need to use in this machine and figure out their imports. The absolutely necessary packages for Heron to run on a non GUI machine are numpy, pandas, pyzmq, paramiko and tornado.

Tornado

After you have installed everything there is a possibility that pyzmq will issue a warning regarding tornado every time you try to run Heron. If this is the case uninstall pyzmq and tornado and then install tornado first then pyzmq and finally update pyzmq (conda update pyzmq).

OpenCV

If you install OpenCV from conda then everything will break. Again this is true at the time of this writing. Maybe, at some point the OpenCV in conda will not generate an inconsistent environment. We wil see. Until this day nothing bad happens if OpenCV is installed through pip. Do:

```
pip install opencv-python
```

or (but not both!)

```
pip install opencv-contrib-python
```

Heron will work with either version. It is up to you if you need the extra functionality of the contrib version.

2.1.3 OS Compatibility

Heron runs on all systems that its libraries can run on. That means Windows, MacOS, Linux and ARM based systems. It has been tested on Windows (10 and 11), MacOS, Linux (Ubuntu 20.04.6, x64) and Raspberry Pi 4 (Debian GNU/Linux 12 (bookworm), aarch64).

Specific OS issues are:

Windows - OpenSSH

If you are on Windows you will not necessarily have openssh up and running. Heron requires this to work properly irrespective of whether you are going to use the LAN functionality of Heron or not. Here is what Microsoft has to say about this.

You will need both the client and the server. To check that everything has worked properly go to where openssh has generated the .ssh folder (check out your user folder) and see if there is a folder in there called known_hosts. If that exists then Heron will not complain. Also after you have set the whole thing up test it out by sshing somewhere from your Windows machine and from somewhere to your Windows machine (making sure both server and client are working).

Linux (both x86 and ARM)

If you install Heron using pip on a machine running Linux (either a PC or a Raspberry Pi) you will need to give executable privileges to the python scripts in Heron. Go into the top Heron directory and do:

```
$ sudo chmod -R 700 ./Heron
```

The 700 will give you (the user) the minimum required privileges. Other, more permissive privilege combinations are also fine.

Raspberry Pi

Installing on Raspberry pi (again at the time of writing this - March 2024) is a little bit trickier. All Heron required libraries except DearPyGui will install with a simple pip. DearPyGui needs to be compiled. Follow the instructions here. Things are still a little experimental (at least until DearPyGui 2.0) so your mileage may vary.

Once DearPyGui is up and running then Heron can be installed either through a pip command or by installing the individual requirements and then downloading Heron from its github page.

Once up and running, Heron might complain that it cannot find the /$HOME/.ssh/known_hosts file. If this is the case then you will need to make an empty known_hosts in the directory Heron is looking for it. This will not bother your standard ssh installation. If you are planning on using the Heron GUI running on Raspberry Pi to run graphs that connect to Nodes on other machines then you need to setup your ssh so that the known_hosts file resides in /$HOME/.ssh.

2.1.4 Node requirements

The above requirements are for Heron and the Nodes that come bundled together in the Heron repository. The heron-repos holds more Nodes, and in the future there will be many more of them. Each Node has its own imports and the environment that runs the worker script of some Nodes needs to have all the required packages both for the basic Heron functionality and for the Nodes it is running.

2.1.5 Environments

It is not a bad idea to put Heron and its basic needs all in a single environment separate from everything else. On the other hand as long as you keep your environment consistent Heron won't complain. The way Heron operates though allows you to have Nodes that work only in different environments than Heron's and with requirements that would clash with each other and still be used in the same pipeline (again see *Setting up and using multiple machines*).

2.2 Startup

2.2.1 After pip install

If you install Heron through pip then you will get a Heron command to start the GUI. On a command line terminal with the correct environment activated just issue the command

```
Heron
```

and the GUI will start.

2.2.2 After manual install

Heron's GUI is just a Python script so the way to run it is by calling in a command line the following code

```
python directory_path_to_Heron/Heron/gui/editor.py
```

If you have used an environment you need to first activate that. If you are on Windows and you do not want to deal with command lines all the time then make a batch file (e.g. Heron.bat) and put in it whatever you would write on your command line. So if for example you have set up a conda environment called base then put in the batch file this:

```
CALL conda activate base
python directory_path_to_Heron\Heron\gui\editor.py
```

If you are on Linux the assumption is you do not need this manual to set up a bash file.

CHAPTER

THREE

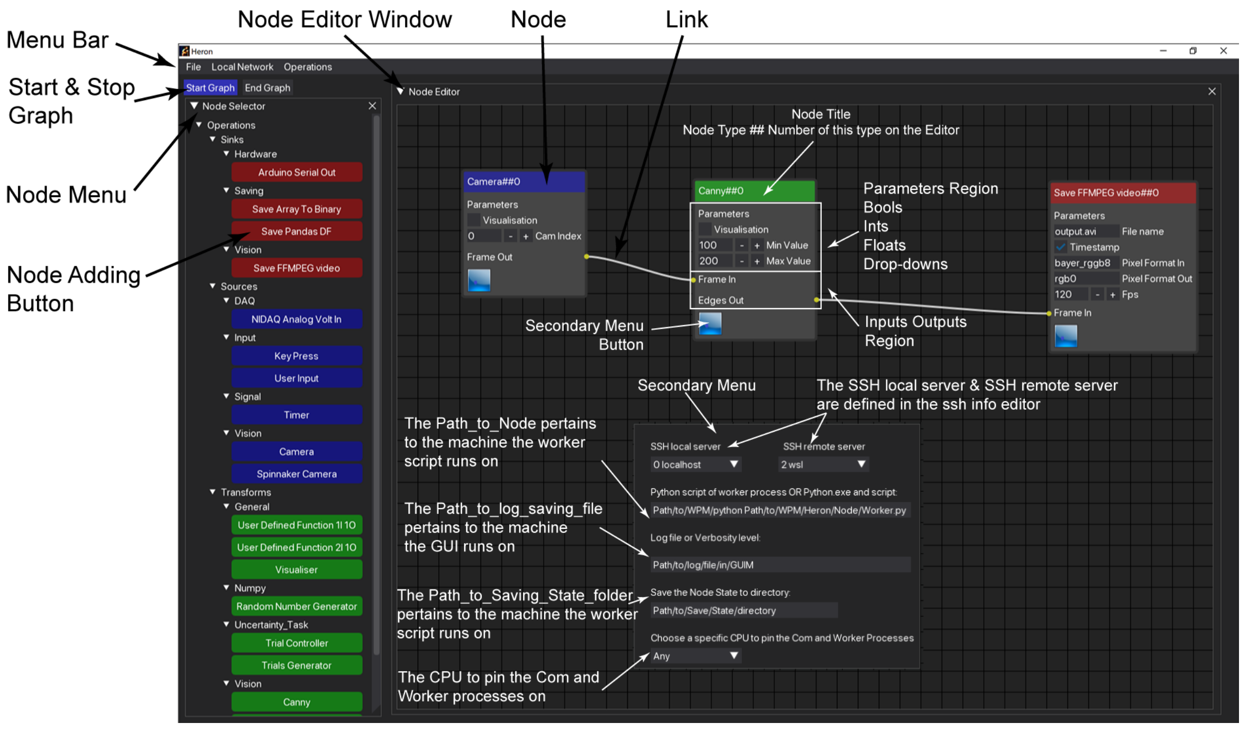
THE EDITOR

3.1 Node Menu and Node Editor Window

Heron's editor is one of the two hearts of the framework (yes, Heron is a beast with two hearts, the other being whatever tool one uses to write the code for the different Nodes).

The editor (see Figure 1) has three main regions, the Menu Bar, the Node Menu and the Node Editor Window (NEW). The Node Menu is where all the Nodes available to Heron are represented as buttons. Pressing one of these will add the corresponding Node to the NEW. The Node Menu is populated automatically by reading the Operations folder of the Heron code base.

The NEW is where a user, having loaded all required Nodes connects them with Links and sets the parameters in them. The parameters are live, that means that when the Graph is running (that is, all the Nodes are running their com and worker scripts and the GUI cannot be used as an Editor), a change in the parameters in the GUI will propagate all the way to the worker script (even on another machine). It is up to the worker script of each Node to decide if it wants to do something with that (see *Writing new Nodes*)



The SSH local server & SSH remote server are defined in the ssh info editor

The Path_to_Node pertains to the machine the worker script runs on

The Path_to_log_saving_file pertains to the machine the GUI runs on

The Path_to_Saving_State_folder pertains to the machine the worker script runs on

The CPU to pin the Com and Worker processes on

SSH local server: 0 localhost

SSH remote server: 2 wsl

Python script of worker process OR Python.exe and script: Path/to/WPM/python Path/to/WPM/Heron/Node/Worker.py

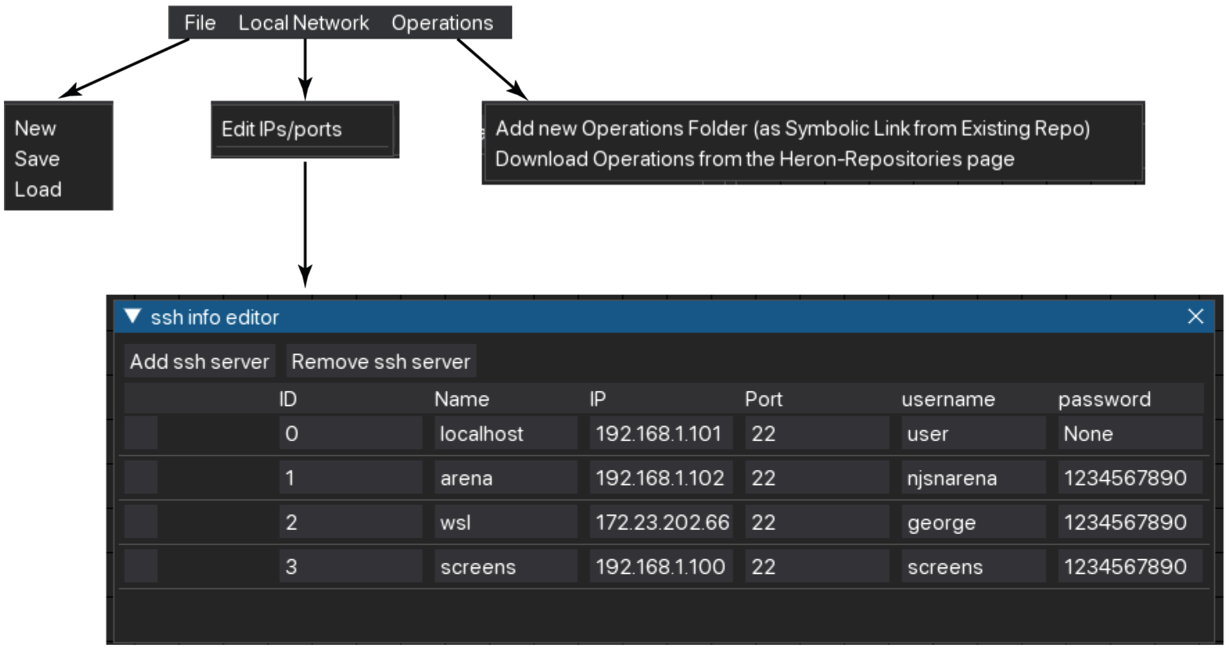
Log file or Verbosity level: Path/to/log/file/in/GUIM

Save the Node State to directory: Path/to/Save/State/directory

Choose a specific CPU to pin the Com and Worker Processes: Any

3.2 Menu Bar

Finally there is the Menu Bar with three drop down menus (see Figure 2). The *File* menu offers the *New*, *Load* and *Save* buttons that do exactly what you'd expect them to do. The *Local Network* menu offers a single item, *Edit IPs/ports*. When this is pressed Heron's SSH Info Editor (SIE) window appears where one can specify the different machines on the network accessible to Heron (see *Setting up and using multiple machines*). Finally the *Operations* button offers the options *Add new Operations Folder (as Symbolic Link from Existing Repo)* and *Download Operations from the Heron-Repositories page*. The second option does not have any functionality implemented yet (i.e. it does nothing at the moment). The first option allows one to point to a repository on their hard disk that is structured to hold code for one or more Heron Nodes and add a symbolic link of this code to the appropriate place in the Heron/Heron/Operations folder so that Heron's Node Menu will see them as valid Nodes (see *Adding Repositories as new Nodes*).


File
Local Network
Operations
New
Save
Load
Edit IPs/ports
Add new Operations Folder (as Symbolic Link from Existing Repo)
Download Operations from the Heron-Repositories page
ssh info editor
Add ssh server
Remove ssh server

ID	Name	IP	Port	username	password
0	localhost	192.168.1.101	22	user	None
1	arena	192.168.1.102	22	njsnarena	1234567890
2	wsl	172.23.202.66	22	george	1234567890
3	screens	192.168.1.100	22	screens	1234567890

3.3 Secondary Node Window

As can be seen in Figure 1 each Node has a blue button at the bottom left that when pressed calls a secondary window allowing the input of extra information regarding the Node. Here is where the user tells the Node:

1. On which machine to run
2. Which worker script to call
3. Where to save log info that comes from the com script of the Node
4. Where to save the Relic if the worker script is implementing one
5. Which CPUs to run the Com and Worker processes of the Node

For points 1 and 2 see *Setting up and using multiple machines*, for point 3 see *All about the Nodes* and *Debugging*, for point 4 see *The Saving State System* and for point 5 see *All about the Nodes*.

3.4 The Graph Control Buttons and the Nodes' life cycle

Under the Menu Bar and over the Node Menu are two buttons called *Start Graph* and *End Graph*. When the Editor is in Edit mode the *Start Graph* is blue (can be pressed) while the *Stop Graph* is grey (cannot be pressed). When the *Start Graph* is pressed the Editor leaves Edit mode and enters Run mode. That means the following:

1. The Node Menu becomes inactive and no new Nodes can be added to the NEW.
2. Heron starts one by one the Nodes in the NEW (in the order they were added to the NEW). As each Node has both its com and worker scripts executed it will show on the NEW a white surround box (indicating both scripts are executing correctly).
3. When all Nodes are running, then the *Start Graph* button will grey out (become inactive) and the *End Graph* one will become blue (able to be pressed).

When the *End Graph* is pressed Heron will close down all com scripts. This shutting down will trigger the end of life system of Heron which a few seconds later will cause all of the worker scripts to self destruct. During that time Heron will inform the user by showing a graphical timer (a download bar).

3.4.1 The closing down period

The number of seconds it takes for the worker scripts to close themselves down after Heron has terminated all of the com scripts (either when the Graph exits or when Heron closes) is defined by the HEARTBEAT_RATE and HEARTBEATS_TO_DEATH parameters found in the constants.py script. By default the HEARTBEAT_RATE is set to one (second) and the HEARTBEATS_TO_DEATH to 5. That means it takes the worker scripts 5 seconds of Heron inactivity to close down.

There is a balance to be had here. If there are worker scripts that need more time to execute a single loop (data entering, data manipulated, data leaving) than the number of seconds it takes for a worker script to close itself down then the script will terminate while it is still doing work. So if you have Nodes killing themselves off without having the chance to do anything check out the time it takes for them to go through one iteration and if it is bigger than HEARTBEAT_RATE * HEARTBEATS_TO_DEATH (in seconds) then increase these constants to give your Nodes time to finish.

On the other hand if all your Nodes do their work much faster than this waiting time then decreasing it will make Heron get out of Run mode and back into Edit mode faster.

The HEARTBEAT_RATE and HEARTBEATS_TO_DEATH constants are local to each machine. So it is feasible that your GUI running machine has a number (say the default 5s) but on another machine you have set this number to another number (say 10s) because the Node on this machine takes its time to complete an iteration. That means though that Heron's GUI will come out of Run mode in 5s but there will still be another 5s that the 2nd machine will need to kill its worker script. If you start the Graph again within those 5s then guess what will happen.

CHAPTER

FOUR

SETTING UP AND USING MULTIPLE MACHINES

One of Heron's major strengths is that it offers multiple machine connectivity. That means that a single pipeline as represented by a Graph on Heron's GUI can be composed by Nodes whose worker processes run on different machines. The following explains how one can setup the different machines to achieve these multi machine pipelines.

4.1 Setting up the machines

In most cases that involve experimental set ups it is the experimenter that will have to physically connect together the different machines in order to create a Local Access Network (LAN). How to do this is a bit beyond the scope of this documentation. Of course if you are lucky and someone else has set up a LAN for you then congrats.

In both cases there is a conceptual separation between the machine that runs Heron's GUI (let's call it the GUI Machine := GUIM) and the machines that simply run one or more worker processes (let's call those the worker process machines := WPMs). The end result of any type of LAN setup should be something that fulfills the following points:

1. All machines should be running an SSH client and an SSH server.
2. The GUIM needs to be able to SSH into all the WPMs (accounts, IPs and ports all working).
3. All WPMs need to be able to SSH into the GUIM (the GUIM account, IP and ports all set up).
4. All WPM SSH accounts (the accounts that the GUIM would use to SSH into the WPMs) need to have enough rights to allow running the worker processes under consideration from an SSH terminal.
5. (Optional but highly desirable) All machines would be nice to have static IP addresses. If they don't and the IPs change every so often then the new IP addresses would need to be updated into Heron's ssh info window. Not fun.
6. All WPMs should have Heron and the required code in the Operations folder of the Heron/Heron folder to run the required worker processes.
7. The GUIM should of course have Heron but should also have all the code in its Heron/Heron/Operations folder for all the Nodes in the pipeline including the Nodes that run on the WPMs.

To give an example of the above points. Let's say that one needs to run a Heron pipeline that needs to control a Near-Zero gimbal motor controller connected to a rapsbery pi. The GUIM needs to have Heron and also the Near-Zero Controller repository installed in Heron's Operations folder (on how to do this see *Adding Repositories as new Nodes*). The WPM (the raspberry pi) needs to also have Heron and the Near-Zero Controller repository in that Heron's Operations folder. Both the GUIM and the WPM need to be running an SSH server and client. Also the account on the WPM that the GUIM will SSH needs to have access rights to be able to run the

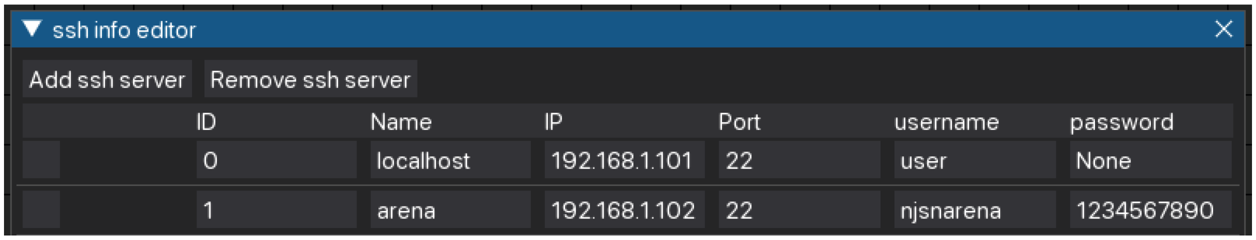
```
path_to_WPM_python.exe path_to_near_zero_controller_worker.py
```

command that the GUIM's Heron will call through SSH on the WPM.

So, setting up a Heron pipeline over multiple machines requires a physical LAN having Heron and all required Nodes on all machines and setting up the SSH clients and servers on these machines in such a way that the GUIM can call python scripts on the WPMs.

4.2 Setting up the LAN in Heron

Once all machines have been connected and configured appropriately then Heron needs to be told about them. This is done through the ssh info window under the Local Network -> Edit IPs/ports menu buttons of the Menu Bar (see *The Editor* and Figure 1). That window is a table where one can input the SSH configuration of all the computers in the pipeline. The *Add ssh server* button will add another entry in the table while the *Remove ssh server* will delete any entry that is checked (with the checkbox at the left of the entry).



	ID	Name	IP	Port	username	password
	0	localhost	192.168.1.101	22	user	None
	1	arena	192.168.1.102	22	njsnarena	1234567890

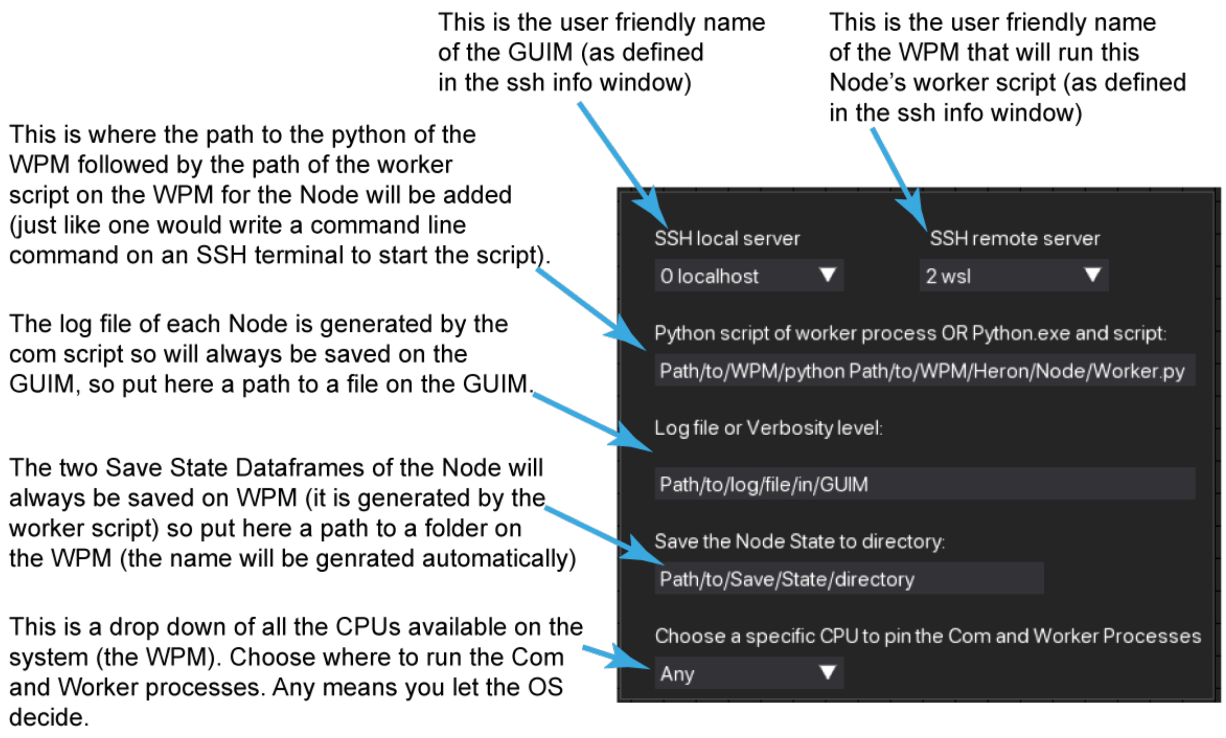
Figure 1.

Each entry has 6 required boxes. The *ID* gets filled in automatically as a new entry is added. The *Name* is a user friendly name of the computer that the user will see in the drop downs where different machines can be added (see further on). The *IP* and *port* boxes are the IP of the machine and the open port that another machine can SSH into. The *username* and *password* boxes are self explanatory. Currently Heron uses a username / password authentication technique. The username and password (as mentioned above) need to belong to an account with enough rights to run the Node Python scripts.

If multi machine pipelines are going to be used there must always be an entry for the local computer (the GUIM). The IP and port for that needs to be what a WPM would use to SSH into the GUIM. There is no requirement for username and password but the boxes need to be filled in with something.

4.3 Setting up the pipeline

The first step to set up any pipeline is to put the required Nodes in Heron's Node editor and connect them appropriately. For a pipeline that has Nodes (worker scripts) running on machines other than the GUIM there is also the requirement to let Heron (the one running on the GUIM) know where the pythons and worker scripts of the WPMs' Nodes are in those WPMs. This is done through the secondary window of a Node (see Figure 2) which appears when the blue button at the bottom left of every Node is pressed..


This is the user friendly name of the GUIM (as defined in the ssh info window)
This is the user friendly name of the WPM that will run this Node's worker script (as defined in the ssh info window)
This is where the path to the python of the WPM followed by the path of the worker script on the WPM for the Node will be added (just like one would write a command line command on an SSH terminal to start the script).
The log file of each Node is generated by the com script so will always be saved on the GUIM, so put here a path to a file on the GUIM.
The two Save State Dataframes of the Node will always be saved on WPM (it is generated by the worker script) so put here a path to a folder on the WPM (the name will be genrated automatically)
This is a drop down of all the CPUs available on the system (the WPM). Choose where to run the Com and Worker processes. Any means you let the OS decide.

SSH local server: 0 localhost
SSH remote server: 2 wsl
Python script of worker process OR Python.exe and script: Path/to/WPM/python Path/to/WPM/Heron/Node/Worker.py
Log file or Verbosity level: Path/to/log/file/in/GUIM
Save the Node State to directory: Path/to/Save/State/directory
Choose a specific CPU to pin the Com and Worker Processes: Any

Figure 2.

The SSH local server has to be populated with the user friendly name of the local computer (as set in the ssh info window) and the SSH remote server with the user friendly name of the WPM that will run the worker script. These are drop down menus and will give as options all the user friendly names currently saved in Heron. The *python script of worker process OR Python exe and script* is automatically populated with the path on the GUIM of the worker script of the Node (so when a Node runs on the GUIM it usually does not need to be changed). But when the Node runs on a WPM then the info here needs to be different. In this case there needs to be the python executable of the WPM followed by the python worker script of the Node on the WPM (both with full, absolute paths). So in the case of the raspberry pi above running a Near-Zero Controller Node this entry would look something like

```
/pi/Miniconda3/python.exe /pi/Heron/Heron/Operations/Sinks/Motion/NearZero_Controller/
↪near_zero_controller_worker.py
```

if for example the python running on the pi was a miniconda installation. Practically one needs to put in this entry box the same command one would need to write in an SSH terminal connected to the base directory of the WPM in order to run the worker script.

The next two entries in the secondary window are by default empty and are used for debugging and data saving purposes (see *Debugging* and *The Saving State System*). The important detail in the case of running a Node in a WPM is that the log file in the *Log File or Verbosity Level* entry box needs to be a file in the GUIM because it is generated by the com process (which always runs on the GUIM). On the other hand the path where the Save State for this Node is going to be saved needs to be a path in the WPM.

The final entry in the secondary window is a drop down menu that allows the user to pin the com and worker processes to a specific CPU. If the value here is set to Any (the default) then the OS chooses how to spread the processes load over the system's CPUs.

Warning: In Windows allowing the OS to define which CPUs are used to run the two processes can lead to pyzmq sending packages from the worker to the com process with latencies of 9 to 50ms! If time accuracy is an issue and you are running Windows then set this to a specific CPU.

4.4 Multiple environments

A direct consequence of being able to define the python executable that a worker process should be run with in each Node's secondary window is the fact that even on the same machine different Nodes can run using different Python environments, each set up specifically to support the functionality of a small set of Nodes. For example the Spinnaker Camera Node can only run under a Python 3.8 environment since the PySpin library it needs to interface with the Flir camera does not run on Pythons newer than 3.8. One can set up a Python 3.8 with PySpin environment separate to the environment that Heron and most of its Nodes run on and still be able to grab the frames a Flir camera generates and process them with code running in different (more modern) environments. All one would have to do is add to the *python script of worker process OR Python exe and script* entry box the full path to the environment's python executable in front of the full path of the worker script to the Node.

CHAPTER

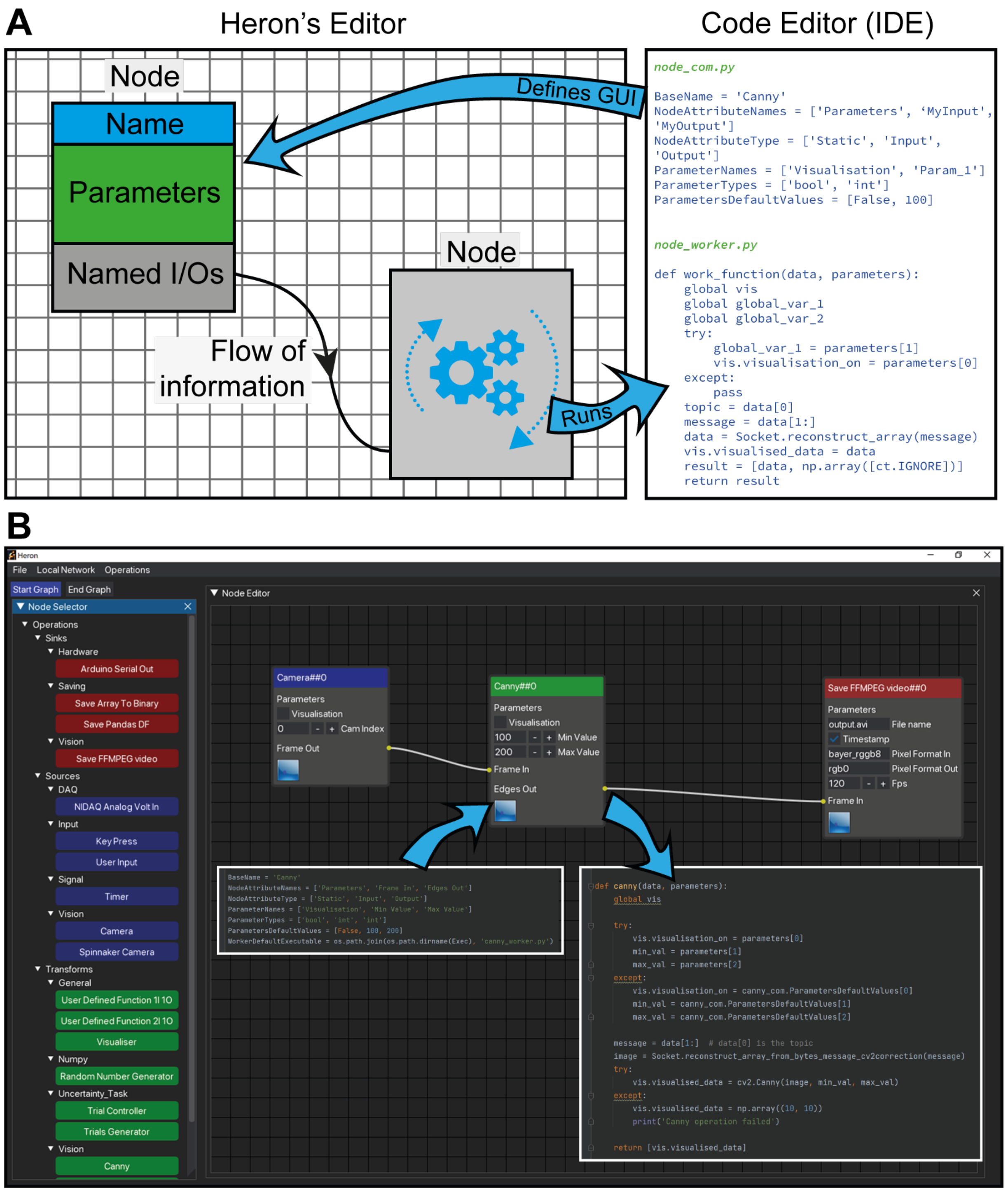
FIVE

ALL ABOUT THE NODES

A Node in Heron is a word of many faces. It is used interchangeably to mean either the two processes that are responsible for executing a certain functionality and communicating with other Nodes (so the Actor in the Actor-based architecture that Heron defines), or the little Node GUI in Heron's Node Editor representing this functionality and giving access to the worker script's state through its Parameters user inputs. It can also be used as a synonym of the folder structure that holds the text based code that Python interprets into this functionality. In most cases the meaning is pretty clear from the context in which the word is used.

So a Node is an amalgam of a group of graphical elements, a corresponding folder structure, a set of necessary and sufficient Python scripts in that structure and the two processes that run when Python interprets those scripts.

The connection between the graphical elements representing a Node, the Node's scripts and the processes that are the functional part of the Node can be seen in Figure 1.



Camera##0: 0 Cam Index

Canny##0: 100 Min Value; 200 Max Value

Save FFMPEG video##0: output.avi File name; bayer_rggb8 Pixel Format In; rgb0 Pixel Format Out; 120 Fps

node_com.py

```
BaseName = 'Canny'
NodeAttributeNames = ['Parameters', 'MyInput',
'MyOutput']
NodeAttributeType = ['Static', 'Input',
'Output']
ParameterNames = ['Visualisation', 'Param_1']
ParameterTypes = ['bool', 'int']
ParametersDefaultValues = [False, 100]
```

node_worker.py

```
def work_function(data, parameters):
    global vis
    global global_var_1
    global global_var_2
    try:
        global_var_1 = parameters[1]
        vis.visualisation_on = parameters[0]
    except:
        pass
    topic = data[0]
    message = data[1:]
    data = Socket.reconstruct_array(message)
    vis.visualised_data = data
    result = [data, np.array([ct.IGNORE])]
    return result
```

```
BaseName = 'Canny'
NodeAttributeNames = ['Parameters', 'Frame In', 'Edges Out']
NodeAttributeType = ['Static', 'Input', 'Output']
ParameterNames = ['Visualisation', 'Min Value', 'Max Value']
ParameterTypes = ['bool', 'int', 'int']
ParametersDefaultValues = [False, 100, 200]
WorkerDefaultExecutable = os.path.join(os.path.dirname(Exec), 'canny_worker.py')
```

```
def canny(data, parameters):
    global vis

    try:
        vis.visualisation_on = parameters[0]
        min_val = parameters[1]
        max_val = parameters[2]
    except:
        vis.visualisation_on = canny_com.ParametersDefaultValues[0]
        min_val = canny_com.ParametersDefaultValues[1]
        max_val = canny_com.ParametersDefaultValues[2]

    message = data[1:]  # data[0] is the topic
    image = Socket.reconstruct_array_from_bytes_message_cv2correction(message)
    try:
        vis.visualised_data = cv2.Canny(image, min_val, max_val)
    except:
        vis.visualised_data = np.array((10, 10))
        print('Canny operation failed')

    return [vis.visualised_data]
```

Figure 1. A. A schematic describing the basic idea of the correspondences between a Node's GUI, the com script that defines this, the Node's functionality / processes and the worker script that defines those. B. A simple example of a Heron pipeline showing fragments of code that define the Canny Node (both its GUI and its functionality).

5.1 The scripts

A thorough description of a Node's folder structure can be found in *Adding Repositories as new Nodes*. Here we will explain what is inside a Node's base folder (usually named using the Node's name) and how the files found there work together to create the Node's GUI and functionality.

Heron expects to see two scripts in the base Node folder. One named *whatever_you_want_com.py* and one *whatever_you_want_worker.py*. It is good practice to use the name of the Node to name those two files but Heron won't mind. Other users of your Node might though. All that Heron will look for is for two python scripts that finish in _com.py and _worker.py. When Heron starts, it scans its Operations folder and creates Node buttons (see *The Editor*) for all folders in the subdirectories inside the Source, Transform and Sink folders assuming that each one of these Node folders has a xxx_com.py and a xxx_worker.py script. The xxx_com.py script will help Heron create the Node's GUI when a Node button is pressed and a Node is added to the Node Editor window. The xxx_worker.py will be called by Heron when a Graph has started.

The _com script is heavily prescribed by Heron and a Node developer is allowed to only change the values of the predefined variables in it. On the other hand the xxx_worker.py script is there for the Node developer to complete and Heron only expects two required functions, the one to be defined as the worker_function and the one to be defined as the end_of_life function. Heron can also deal with one optional function, to be defined as the initialisation function. For more details on how Nodes are composed through the xxx_com.py and xxx_worker.py scripts see *Writing new Nodes*.

5.2 The Node types

Heron understands three types of Nodes, Sources, Transforms and Sinks. The main difference between them is the inputs and outputs they are allowed to have. Sources generate their own data and can only have outputs (no inputs). Transforms take data from other Nodes into their inputs and, well, transform them, sending the transformed data out to other Nodes through their outputs. So they have both inputs and outputs. Finally Sinks are meant to take data from previous Nodes in the pipeline and do something with them without passing them further on (so, no outputs).

5.3 Creating a Graph (Pipeline)

To create a pipeline ones does the pretty obvious. Presses the Node buttons that the pipeline requires, connects the Nodes that have now appeared on the Node Editor window (by connecting an output of one Node to an input of another) and finally sets the parameter values for the different Nodes (don't forget to save). That's it. Well, there is also the issue of telling the Nodes whose worker function is to run on another machine, where exactly that machine is (done through the Node's secondary window as described in *The Editor* and *Setting up and using multiple machines*). But now that is it for real.

5.4 Running a Graph (a Node's life)

Once a Graph has been constructed then one can press the *Start Graph* button which will do two things. First it will stop Heron from acting like a pipeline editor (no Nodes can be added, deleted or (dis)connected). Second it will go through all the Nodes in the Node Editor window and (in order of addition to the window) it will start their respective xxx_worker.py scripts. That means that Heron will first check to see if an initialisation function is defined in the xxx_worker.py script and if yes it will call it, and keep calling it until it returns True. If there is no initialisation function or when it has returned True, Heron will pass the parameter values from the Node's GUI to the xxx_worker.py script and then, well it depends on what type the Node is.

For Source Nodes, Heron will call the worker_function in the xxx_worker.py script once. This assumes that the function is an infinite loop that generates data and deals itself with passing those to the com process of the Node (this is much simpler than it sounds and you can see how it is done in *Writing new Nodes*).

For Transform and Sink Nodes Heron will call the worker_function of the xxx_worker.py script as a callback every time that new data have arrived in the Nodes inputs. The worker_function of these Nodes is expected to return a list of numpy arrays, each array corresponding to one of the Node's outputs, which Heron will deal with from that point on (again see *Writing new Nodes*).

Heron will call the worker_function of a Node if there are new data in AND the previous call has returned. If any new data arrive at the Node's input and the worker_function from the previous call is still running then Heron will drop the new data which will be lost.

Warning: Heron has no buffer to hold any messages that come into a Node that is still processing its previous message. This is by design.

The logic behind the no-buffer feature is because in Heron's use cases there is no situation where a Node would receive large amounts of data in bursts and very little data during the rest of the time (in which case a buffer would make sense). Nodes in most experiments will either be data intensive but with a constant or near constant data receiving speed (e.g. cameras) or will have variable data load reception but always with small data loads (e.g. buttons). The second case is not an issue and the first case cannot be dealt with a buffer but with the appropriate code design, since buffering data coming in a Node too slow for its input will just postpone the inevitable crash.

This design approach though means that Nodes can and will drop incoming packets silently. This is why Heron does not rely on packet order to tell which incoming packet in a Node corresponds to a Node's state change or outgoing packet. To achieve this important time matching functionality Heron offers a number of debugging (see *Debugging*) and saving (see *The Saving State System*) tools that allow the user to know exactly which incoming packets never reached the Node and which generated packets were dropped by the next Node in the pipeline. This is described in more detail in *Synchronisation*.

The above fully defines what a Node does during an active Graph. Once the *Stop Graph* button has been pressed (or Heron is closed down) then all processes (the three forwarders and the com and worker processes for all Nodes) are killed (see below). At this point each Node will execute its on_end_of_life function taking care of any loose ends.

5.5 The Heartbeat System

While a Graph is running Heron sends every so often a message to all the Nodes telling them that all is fine and they should keep on operating. This is called a Heartbeat and the number of heartbeats per second is defined in the constants.py script (Heron/Heron/constants.py) as HEARTBEAT_RATE. It is by default set to 1 (message per second) but it can be changed by the user if needs be.

Each Node while running is also running on a separate thread a receiver of the heartbeat message. This thread keeps track of how long ago the last received heartbeat message was delivered to the Node. If this time surpasses HEARTBEATS_TO_DEATH seconds (also defined in the constants.py script) then the worker process of the Node calls its on_end_of_life function and then terminates itself.

This system allows the worker process that is running on a different machine to actually terminate without the com process that initiated it in the first place having to issue a kill command (which in the case of processes over different machines usually doesn't work).

The HEARTBEATS_TO_DEATH is an important constant for a user to control (and eventually will be accessed through Heron's GUI). It tells each Node how long it should wait without a heartbeat signal before it kills itself. There is a very important caveat to this system. In a Transform or Sink Node a heartbeat can be received only every time the worker_function is called (a Source's heartbeat is received any time it is sent). If it takes the worker process to long to either initiate (e.g. because of a slow connection between machines when the Node's worker process is running on a different machine) or the worker_function to actually run a single time then the worker process will terminate itself since the check of the time passed since the previous received heartbeat is done continuously. That means that for Transfer and Sink Nodes that are slower in their initialisation or worker_functions than HEARTBEATS_TO_DEATH * HEARTBEAT_RATE seconds the Node will always kill itself. The solution to this is to increase one of those two constants. These constants can have different values on different machines.

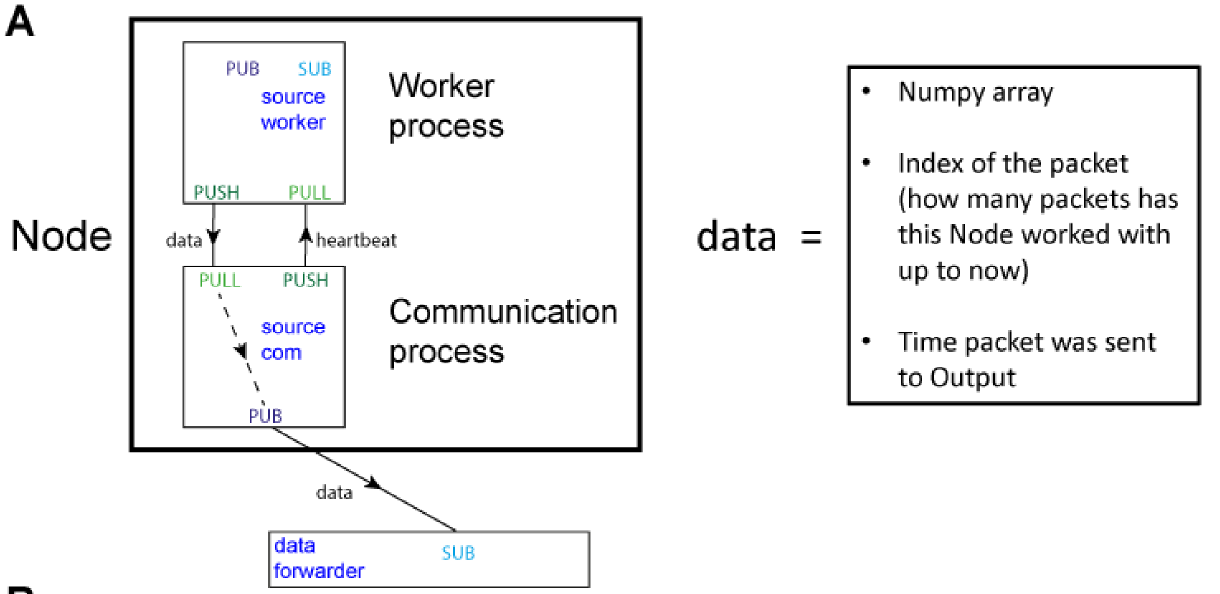
5.6 The inner workings

When a Graph is starting, the Heron GUI process will go through each of the Nodes in the Graph and will spun up a process. This is known as the com process of the Node. This always runs on the same machine as the Heron GUI and is responsible for the communication between Nodes. When the com process is up and running it will start a second process called the worker process of the Node. This runs the code that is in the xxx_worker.py script of the Node. The worker process will do the main work of the Node and will communicate with the com process in order to get any messages from other Nodes or send messages to other Nodes. The messaging between com processes is facilitated through another process called data forwarder and is done using the PUB SUB protocol of 0MQ.

There are two more communication pathways to be considered that deal with the communication between Heron's GUI process and each of the Nodes' worker processes. The first pathway deals with an initial messaging between the worker process and the Heron GUI process and is required so that the GUI process becomes aware when the worker process is up and running. The second allows the GUI process to send messages to the worker processes every time a parameter in the Nodes' GUIs changes. Currently Heron does not allow the worker process of a Node to change the parameters shown on the GUI of the Node.

A schematic of all the processes and communication pathways between them can be seen in Figure 2B.

A



B

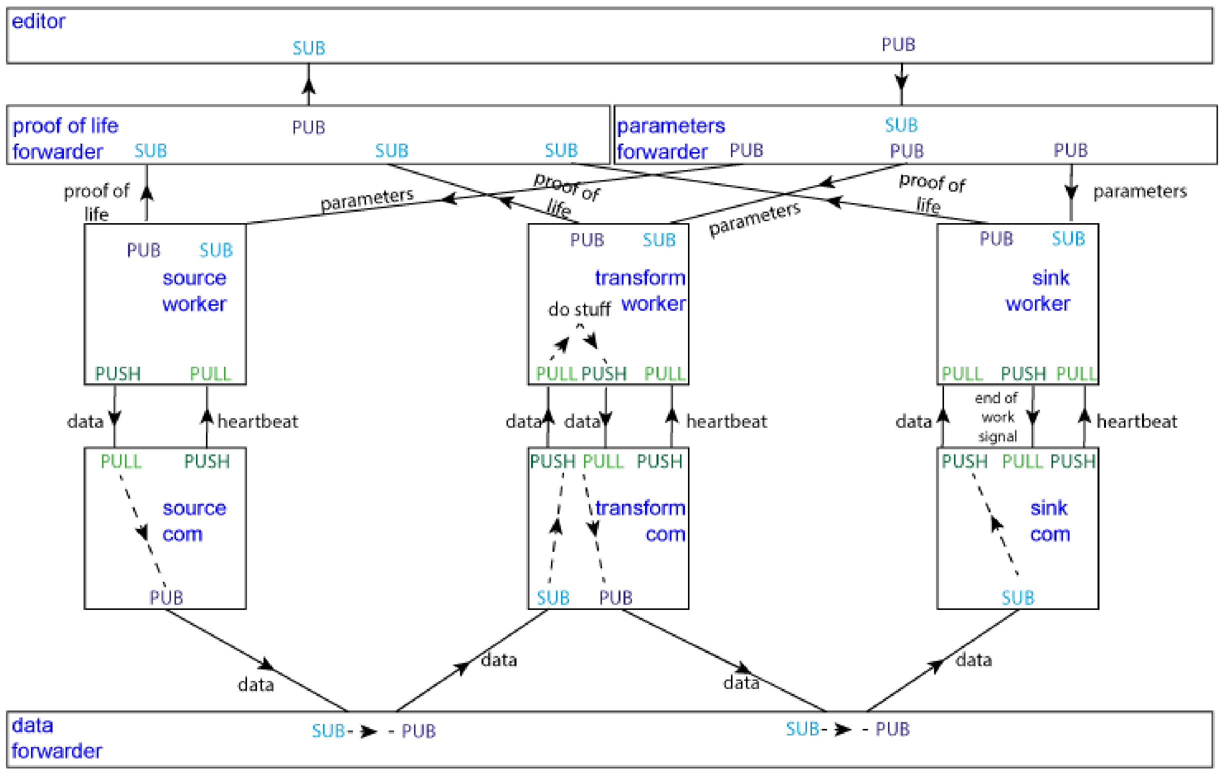


Figure 2. Heron's design principles. A. On the right, Heron's Node structure showing the two processes that define a running Node and how messages are passed between them. The design of the special message defined as 'data' is shown on the left. B. Heron's processes and message passing diagram. Each rectangle represents a process. The top rectangle is the Heron process, while the two rectangles below that and the rectangle at the bottom of the figure are the three forwarders. The proof of life forwarder deals with messages relating with whether a worker process has initialised correctly. The parameters forwarder deals with messages that pass the different parameter values from the editor process (the GUI) to the individual Nodes. Finally, the data forwarder deals with the passing of the data messages between the com processes of the Nodes. The squares represent the worker (top) and com (bottom) process of three

Nodes (a Source, a Transform and a Sink from left to right). Solid arrows represent message passing between processes using either the PUB SUB or the PUSH PULL type of sockets as defined in the 0MQ protocol. Dashed line arrows represent data passing within a single process.

5.7 Choosing the CPU core

Heron allows the user to choose the CPU a worker process of a Node will run on. The default is to allow the system to choose. This is not best practice though and often not constraining the worker process to a specific CPU will lead to increased overall resource usage and more importantly (at least in Windows) will also lead to a large amount of dropped packets.

Warning: If a Node seems to be dropping packages in either sending and receiving them, one of the first possible things to try (before much heavier code optimisation) is to lock the CPU the node runs in to a specific core. This often mitigates the problem, especially in Nodes that receive packages (Transforms, Sinks and Nodes that are running non Python code as a separate process). CPU locking though is not a panacea when it comes to package dropping and in certain cases might result in an increase of package dropping, so test your system and use appropriately.

CHAPTER SIX

ADDING REPOSITORIES AS NEW NODES

6.1 Adding an existing repository

Heron can add repositories that have the correct folder structure (see below) as symbolic links into its own folder structure and thus allow Heron's GUI to see code in these repositories as Nodes. This is the principal way of adding new Nodes to Heron's GUI.

The first step in that process is to create or download a repository designed to be a Heron Node or group of Nodes. Once such a repository is in place on the hard disk somewhere (do not put it in Heron's folder structure) then do Menu Bar -> Operations -> Add new Operations Folder (as Symbolic Link from Existing Repo). Point Heron to the top folder of the repository and it will add the correct parts of it in Heron's Operations folder as symbolic links.

In Windows, in order for an app to make symbolic links it needs to have Administrator rights, so to do the above in Windows, Heron must have been started with a Run as administrator (Linux does not require anything like this). If the above is not the case then Heron will give a warning reminding the user to give it elevated privileges. Another error will be generated if the directory Heron is pointed to does not contain a valid Heron Nodes folder structure.

6.2 Downloading a repository from inside Heron

Currently this functionality (Menu Bar -> Operations -> Download Operations from the Heron-Repositories page) is not implemented. In the future you will be able to download a repository straight from a repository website and into a new repository on your machine with the symbolic links taken care of.

6.3 Creating a valid Heron Nodes repository from scratch

The only requirement for a valid Heron Node repository is its correct folder structure. This is as follows

```
Base_repository_folder
    Sources / Transforms / Sinks
        Subcategory
            __top__
                ignore.gitignore
            Name_of_Node
                com_script.py
                worker_script.py
```

Here is an example with 4 Nodes in it, a Source called *Weird Camera* (under Vision), a Transform called *Super Motor Controller* (under Motion) and two Sinks called *Saving CSVs* (under Saving) and *Some Other Sink Node* (under General).

```
My_awesome_Nodes_repo
    README.md
    Sources
        Vision
            __top__
                ignore.gitignore
            Weird_Camera
                weird_camera_com.py
                weird_camera_worker.com
    Transforms
        Motion
            __top__
                ignore.gitignore
            Super_Motor_Controller
                Maybe_Another_Folder
                    another_script.py
                super_motor_controller_com.py
                super_motro_controller_worker.py
    Sinks
        Saving
            __top__
                ignore.gitignore
            Saving_CSVs
                some_other_script.py
                saving_csvs_com.py
                saving_csvs_worker.py
        General
            __top__
                ignore.gitignore
            Some_Other_Sink_Node
                some_other_sink_node_com.py
                some_other_sink_node_worker.py
```

The actual name of the Node as seen on Heron's GUI is set in the xxx_com.py script. It is good practice to give the folder holding the Node (i.e. the Python scripts) the same name but with underscores instead of spaces. Heron doesn't mind if the name of the folder is different but it does mind spaces so pretend it is 1980 and make folder names without spaces.

The folder __top__ needs to exist if the Subcategory you are making under the Sources/Transforms/Sinks folder doesn't already exist in your Heron folder structure. Otherwise it can be skipped. In general it is a good idea to just include it. The ignore.gitignore file is an empty file that gets added to the git repository so that the __top__ folder is also added to the repository.

The file / folder structure inside the Node folder can be anything as long as the xxx_com.py and xxx_worker.py scripts are also present.

The Heron repos organisation on GitHub has plenty of examples of Node repositories.

CHAPTER

SEVEN

WRITING NEW NODES

In order to write a new Heron Node one needs to do two things. First create a correct Node folder/file structure and then populate the xxx_com.py and xxx_worker.py scripts with the appropriate code.

The first step is fully described in the 'Creating a valid Heron Nodes repository from scratch' paragraph of the *Adding Repositories as new Nodes* section. The one thing that is beyond the scope of this manual, is not necessary, but is most definitely good practice is to create a repository for the new Node(s), (ideally in the Heron-Repositories GitHub organisation, but anywhere is better than nowhere).

This part of the manual will deal with the second step, that of writing the code in the two Python scripts to create a functional Node.

7.1 The Node templates

In order to get started with Node writing, the fastest way is to copy into the new Node folder the xxx_com.py and xxx_worker.py template scripts from Heron's templates directory according to what type of Node you are trying to construct (Source, Transform or Sink). The xxx_com.py scripts for all three types are identical but the xxx_worker.py ones are different. Of course using as a template another Node that is similar in functionality to what you want to achieve is often better than a generic template so also have a look through the code of the existing Heron Nodes in the Heron-Repositories organisation.

In the templates there is code that must not be altered, code that is placeholder for your code and comments clearly marking what is what and giving advice on how to write your own code. If you are new to writing Nodes in Heron read carefully through all the commends of the scripts to familiarise yourself with what you need to do to create a functional Node code base.

Because the template scripts were authored in PyCharm some comments are actually PyCharm editor-fold structures that allow you to minimise and maximize different parts of the scripts. Ignore them if you are not using a JetBrains editor.

7.2 The com script

The xxx_com.py script (from now on referred to as simply the com script) deals with the basic definition of the Node.

This definition comprises of the following variables:

- BaseName
- NodeAttributeNames
- NodeAttributeType

- ParameterNames
- ParameterTypes
- ParametersDefaultValues
- WorkerDefaultExecutable

7.2.1 The Node's Name

The BaseName is a string that provides the name of the Node (e.g. 'Camera' or 'My Super Duper Node'):

```
BaseName = 'My Super Duper Node'
```

7.2.2 The Node's Attributes

Each Node can have three types of Attributes. Its inputs (with Type 'Input'), its outputs (with Type 'Output') and its parameters (with Type 'Static'). The NodeAttributeNames is a list of strings that define the names of the Node's attributes. If the Node has parameters then by convention the first item in the NodeAttributeNames is 'Parameters'. So for example a Transform Node with parameters, two inputs and one output would have its Attributes names defined as follows:

```
NodeAttributeNames = ['Parameters', 'Data A In', 'Data B In', 'Transformed Data Out']
```

If a string of a NodeAttributeName also has the word 'Dict' in it (e.g. 'Transformed Data Out Dict') then the Node's GUI will change the colour of that Input or Output to show to users that this Input is expecting a dictionary or this Output is sending out a dictionary (instead of a numpy array). As mentioned above the type of each Attribute can be 'Static', 'Input' or 'Output'. These are defined in the NodeAttributeType entry again as a list of strings (following the same order as in the NodeAttributeNames list). So for the above example the types would be defined as

```
NodeAttributeType = ['Static', 'Input', 'Input', 'Output']
```

7.2.3 The Node's Parameters

The next three entries (ParameterNames, ParameterTypes and ParametersDefaultValues) are there to fully define the parameters of the Node, if it has some. In case the Node doesn't have any parameters then assign an empty list to all the entries (e.g. ParameterNames = []).

The ParameterNames is a list of strings with the names of the parameters. The ParameterTypes is a list of strings with the type of each parameters (again same order as in the ParameterNames list). The possible types are:

- 'str'
- 'bool'
- 'int',
- 'float'
- 'list'

Each parameter according to its type will generate an appropriate element on the Node's GUI.

The first four types and their GUI elements are self-explanatory. The 'list' type is a parameter with a drop-down element.

The ParametersDefaultValues is a list that defines the default values of each of the parameters (what value each parameter gets when a Node is added to the Node Editor). These values must be of the correct type (so a str for a 'str' type parameter, int for an 'int', etc.) The 'list' parameters use their default values, which is a list of strings, to also define the elements of the drop-down GUI element. The default value of these parameters is the first item in this list.

For example if a Node has two parameters, one 'int' called My Integer and one list called My Dropdown the above variables would look like this

```
ParameterNames = ['My Integer', 'My Dropdown']
ParameterTypes = ['int', 'list']
ParametersDefaultValues = [5, ['1st item', '2nd item']]
```

7.2.4 The Node's worker script

Finally the Node needs to easily find its worker script for the (most) cases where the worker script runs on the same machine as the Heron GUI and under the same Python environment. In this case the user shouldn't need to specify where the script is. This is achieved by the WorkerDefaultExecutable variable. Since the worker script is always in the same directory as the com script the WorkerDefaultExecutable variable should always be defined as follows

```
WorkerDefaultExecutable = os.path.join(os.path.dirname(Exec), 'xxx_worker.py')
```

where 'xxx_worker.py' is the string of the name of the worker script provided by the Node's developer.

7.3 The worker script

The worker script is the script that defines the worker process. It is where the main code of the Node's functionality is written. That doesn't mean that the Node cannot have code defined in more scripts, but the worker script (xxx_wroker.py) is the script that the rest of Heron's communication protocol interacts with.

The worker script like the com script has some code that needs to always exist and is common to all worker scripts. This can be found in the Node templates and is annotated appropriately. The user added code needs to define two functions and can define an optional third one. The optional (but highly recommended) function is the initialisation function. The two required ones are the worker function and the end of life function.

The three functions are defined in the first call in the main if loop of the script. E.g. for Transform Nodes this line would look like:

```
if __name__ == "__main__":
worker_object = gu.start_the_transform_worker_process(work_function=some_work_function,
                                                      end_of_life_function=on_end_of_
↪life,
                                                      initialisation_function=initialise)
```

where the some_work_function becomes the worker function, the on_end_of_life function becomes the end of life function and the initialise becomes the initialisation function.

7.3.1 The worker object

Before continuing with the description of the worker script we need to explain what the worker object (named as worker_object in code) is. Each Node type has two classes that define Heron's basic Node functionality (irrespective of what the Node actually does) and ensure Heron's communication protocol runs properly. One class is called XXXCom and the other XXXWorker where XXX can be Source, Transform or Sink. When the worker process spins up the first thing that happens is the creation of a worker_object of type XXXWorker (e.g. worker_object = SourceWorker(lots of arguments)).

This worker object is the main way the rest of Heron communicates with the worker process. The Node developer can use the worker_object because it gets passed in some of the worker script functions when they are called (by the worker_object itself). Where this is useful will be discussed further down.

7.3.2 The differences between Source Nodes and Transform and Sink Nodes

Because Transform and Sink Nodes operate on incoming data, their worker function is a callback that Heron's communication protocol will automatically call every time a new message arrives into the worker process of the Node. On the other hand, in the case of the Source Nodes, the worker function generates its own data so it needs to be developed as a loop that runs for as long as the Node is running. How to do this is clearly shown in the Source template. So in the case of Transforms and Sinks, Heron will call the worker function every time new data come in, while in the case of Sources, Heron will call the worker function only once at the start of the worker process.

The above difference also generates a second difference that has to do with the communication of the worker processes with the Heron GUI process. This communication allows the parameters set in Heron's GUI to pass to the corresponding worker processes. For more details see the following paragraph on the initialisation function.

Finally this difference means also that the worker functions of the Transforms and Sinks are passed different arguments than the worker functions of the Sources when they are called by the Heron framework. Again for more details keep reading.

7.3.3 The initialisation function

The initialisation function is used in order for the Node to run any initialisation code before it starts calling the worker function. Apart from initialisation code pertaining to the specific Node, all worker processes need to check that they can read the parameters sent from the Heron GUI process. This communication takes some time to initiate during which the worker function must not be called. When a worker process starts, Heron will send the Node's parameters to the worker process. It will try to do so NUMBER_OF_INITIAL_PARAMETERS_UPDATES times (this variable is set in the constants.py script) with 500ms gap in between. If it fails then the worker process will not function and it will terminate after HEARTBEAT_RATE * HEARTBEATS_TO_DEATH seconds. Every time Heron's GUI sends parameters to the worker process, the process checks if it has an initialisation function and if it is marked as initialised. If it has an initialisation function and isn't initialised it will call its initialisation function (which should always try to read the parameters). If it returns True then the parameters have been read (and all other initialisation has been completed) and then the worker process is marked as initialised.

In order for the above mechanism to work the initialisation function must always check if it can read parameters from the worker_object. This is done with code that looks like this

```
try:
    parameters = worker_object.parameters
    global_var_1 = parameters[1]
    global_var_2 = parameters[2]
    global_var_3 = parameters[3]
    global_var_1 = parameters[4]
```

(continues on next page)

(continued from previous page)

```
except:
    return False
```

The worker_object is passed as an argument to the initialisation function.

In the case of Transform and Sinks, every time new data come and before the worker function is called, Heron checks if the worker process has been marked as initialised. If this is not the case the worker function is not called and the incoming message is dropped. Once the process is marked as initialised the worker function gets called normally for every new incoming message.

In the case of Sources the worker function gets called only once so the above mechanism is not applicable. The communication between the Heron GUI and the worker process though still might require a little bit of time to be established and before that happens (and thus the parameter values can be known) the infinite loop of the worker function cannot start. Ensuring that the loop starts after the parameters are properly updated is, in the case of Sources up to the Node's developer. See the worker function paragraph on how this is done.

7.3.4 The worker function

The worker function is where the main code of the Node needs to be constructed. As mentioned above in the case of the Transforms and Sinks this function needs to be a callback while in the case of the Sources the main functionality is an infinite loop.

Sources

The Source Node worker function is passed a single arguments, namely the worker_object we described above. As mentioned above the Source worker function needs to give Heron some time to communicate with the worker process before it starts generating data. This is usually done with a small loop before the infinite loop, which ensures the initialisation function has run properly and the parameters can now be read from Heron's GUI:

```
need_parameters = True

while need_parameters:
if worker_object.initialised:
    need_parameters = False
    running = True
    gu.accurate_delay(10)
```

The worker_object.initialised is how a worker process is marked as initialised or not and it will be true only after the initialisation function returns true.

The worker function of a Source Node does not return anything. In order to push the data generated in every iteration of its infinite loop to the com process of the Node it needs to call the following function:

```
worker_object.send_data_to_com(result)
```

where result is what the Node needs to send on and can be either a numpy array of arbitrary dimensions and type or a json compliant dictionary (i.e. a dictionary that can be saved into a json file without errors).

A current limitation of Heron is that Source Nodes cannot have more that one output (the way Transforms and Sinks do).

Transform and Sinks

The worker function fo the Transform and Sink Nodes get passed two or three arguments (i.e. the developer can implement it with either two or three arguments). The two arguments that get always passed are the parameters (as they are currently displayed on the Node's GUI) and that new data that are responsible for calling the worker function in the first place. The third (optional to implement) argument is a function that allows saving in the Save State System anything the developer wants (see *The Saving State System* for a description of the use of this argument).

The parameters is a list of the current parameter values.

The data is a list of two items. The first is a string that fully describes the Node and output of the Node that sent the data and the Node and input of the Node that is receiving the data (i.e. that is, the current Node and the name of the input from which the data came through). The format of the topic is

previous_node_output_name##previous_node_name##previous_node_index -> this_node_input_name##this_none_name##this_node_i

An example (of a topic that would connect the Frame Out output of a Camera Node to the Frame In input of a Canny Node) would be:

Frame Out##Camera##0 -> Frame In##Canny##0

The topic is useful for the worker function to distinguish between data coming in from different inputs of the Node or from different output Nodes if multiple Nodes are connected to this Node's inputs.

The second part of the data list is the actual payload which consists always of a message that needs a little bit of reconstruction. That is achieved with either the

```
message = Socket.reconstruct_data_from_bytes_message(message)
```

or

```
message = Socket.reconstruct_array_from_bytes_message_cv2correction(message)
```

functions of the Socket class (from Heron.communication.socket_for_serialization import Socket)

The reconstruct_array_from_bytes_message_cv2correction function is used to correct an OpenCV bug that breaks the library if the incoming numpy array has signed data. So use it when dealing with images, or when you want to make sure for some other reason that the numpy array you operate on has unsigned data. The reconstruct_data_from_bytes_message will work with both numpy arrays of arbitrary type and with json compliant dictionaries.

Once the worker function has the topic and the numpy array or dictionary coming into the Node then it can do the work required.

The Transform Nodes also have output. In contrast to the Source Nodes, the worker function of a Transform creates the Node's output simpy by returning a list of numpy arrays and or dictionaries. The list must be as long as the number of outputs defined for the Node (this is done in the com script as shown above). The order of the numpy arrays / dictionaries is the same as the order of the outputs defined in the com script. If a worker function needs to output nothing to one or more of its outputs then it needs to pass the ct.IGNORE string (as defined in the constants script of Heron) but needs to wrap it in a numpy array: np.array([ct.IGNORE]). So for example a Transform Node with two outputs that should return the array my_array on the first and nothing on the second would have a return statement that looks like this:

```
return [my_array, np.array([ct.IGNORE])]
```

If the Node has a single output then the numpy array or dictionary returned still needs to be put in a list:

```
return [my_array]
```

There are two more elements of Node scripting, the *in Node Visualisation API* and the *Save State System for saving state* which are described in their own documentation.

7.3.5 The end of life function

The final function that must be defined in a worker script is the end of life function. Heron will call this function when the process terminates itself (see the Running a Graph (a Node's life) paragraph in *All about the Nodes*). This is where code that deals with gracefully closing down the process should be written (e.g. closing graphical elements, releasing memory, etc.). Since this function has to be defined, if there is nothing to close down then a pass call should be used.

CHAPTER

EIGHT

IN NODE VISUALISATION

8.1 The API

Heron defines an API for simple visualisation that the Node developer can use to add data visualisation elements to a Node without developing them from scratch.

This is the API that is also used by the Visualisation Node.

The in Node visualisation is based on the class VisualisationDPG in the gui/visualisation_dpg.py script.

To use the API one first needs to import the class and define a global variable with VisualisationDPG type:

```
from Heron.gui.visualisation_dpg import VisualisationDPG

vis: VisualisationDPG
```

then, in the initialisation function the vis object needs to be instantiated. The VisualisationDPG class takes 8 arguments:

- _node_name: The node's name (found in worker_object.node_name)
- _node_index: The node's index (found in worker_object.node_index)
- _visualisation_type: This defines the type of visualisation. Currently the VisualisationDPG class supports images, text boxes, plots with a single pane which can show 1D and 2D arrays and plots with multiple panes which can show only 2D arrays. In both the single pane and multi pane plots the x axis of the panes is given by the left (fastest changing) axis of a 2D array. The possible values of the argument can be 'Image', 'Value', 'Single Pane Plot', 'Multi Pane Plot'
- _buffer: The number of data points shown on a single visualisation window if the visualisation_type is a Plot or the number of the latest data messages that have been received or generated by the Node if the visualisation_type is Value. This does nothing if the visualisation_type is an Image.
- _image_size: The size of an image if the _visualisation_type is 'Image'
- _x_axis_label: A string with the name of the x axis of the plot(s)
- _y_axis_base_label: A string with the base name of the y axes for the Multi Pane or the name of the y axis for the Single Pane. In the Multi Pane the actual names will be numbered ('_y_axis_base_label 0', '_y_axis_base_label 1', etc.)
- _base_plot_title: A string giving the base label name for each plot for Multi Pane, or the name of the plot for Single Pane. In the Multi Pane the actual plots will be numbered starting from 0 as in the case of the _y_axis_base_label

An example of making a VisualisationDPG object for a text box type of visualisation is the following:

```
visualisation_type = 'Value'
buffer = 20
vis = VisualisationDPG(_node_name=_worker_object.node_name, _node_index=_worker_object.
↪node_index,
                       _visualisation_type=visualisation_type, _buffer=buffer)
```

In the worker function one needs to first check if the visualisation is visible or not (assuming a Visualisation parameter has been defined in the com script) and then pass the data to be visualised to the vis object:

```
global vis

vis.visualisation_on = parameters[0]

vis.visualise(some_data_to_visualise)
```

The VisualisationDPG object will take care of creating and killing the required separate threads so that the visualising element requested through the visualisation_type argument will show only when the vis.visualisation_on is true.

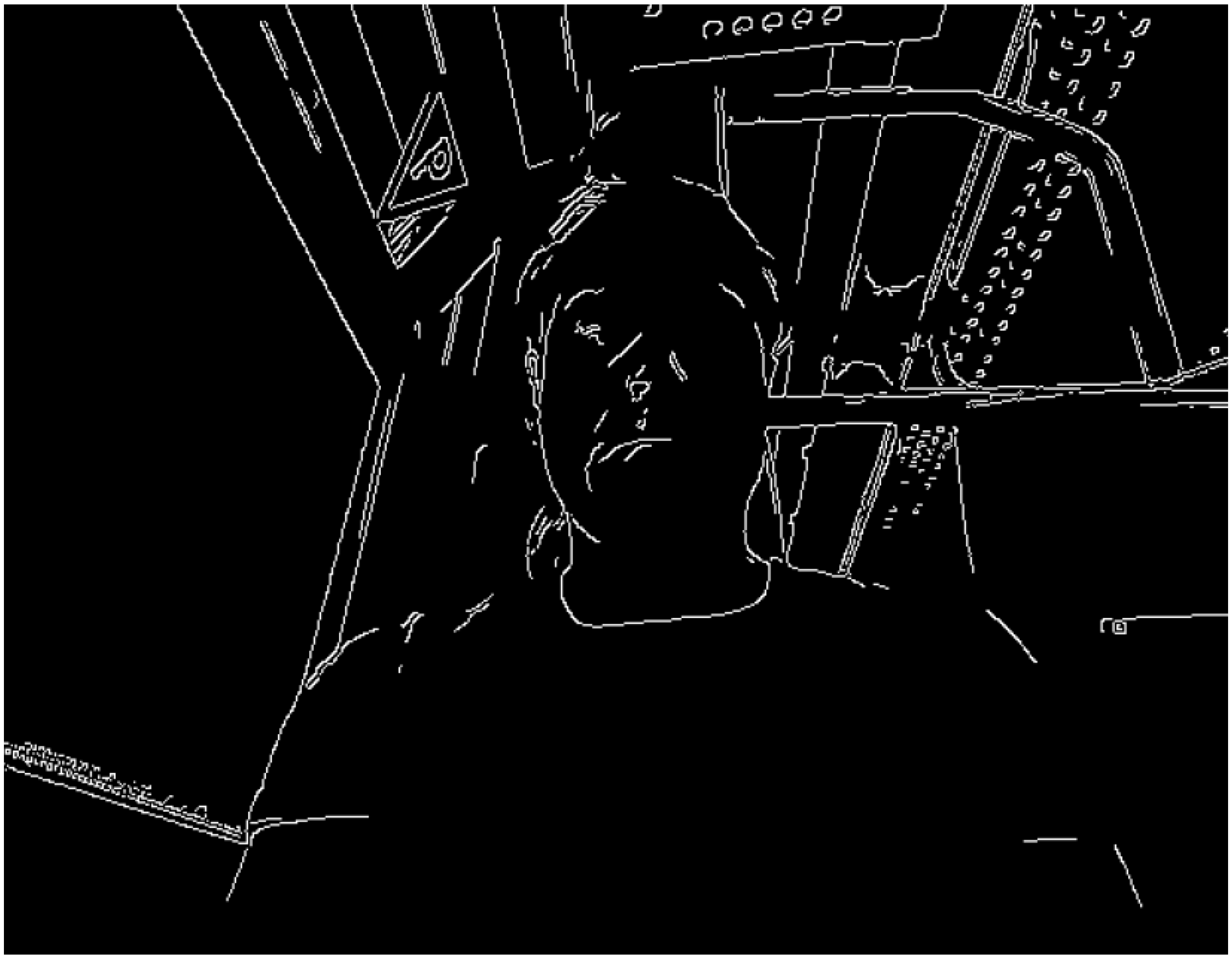
Finally, the VisualisationDPG object needs to properly close down some graphical elements before the process terminates so in the end of life function one needs to add:

```
global vis
vis.end_of_life()
```

Nothing is stopping a Node to have multiple visualisations. Just create multiple VisualisationDPG objects following the above ideas.

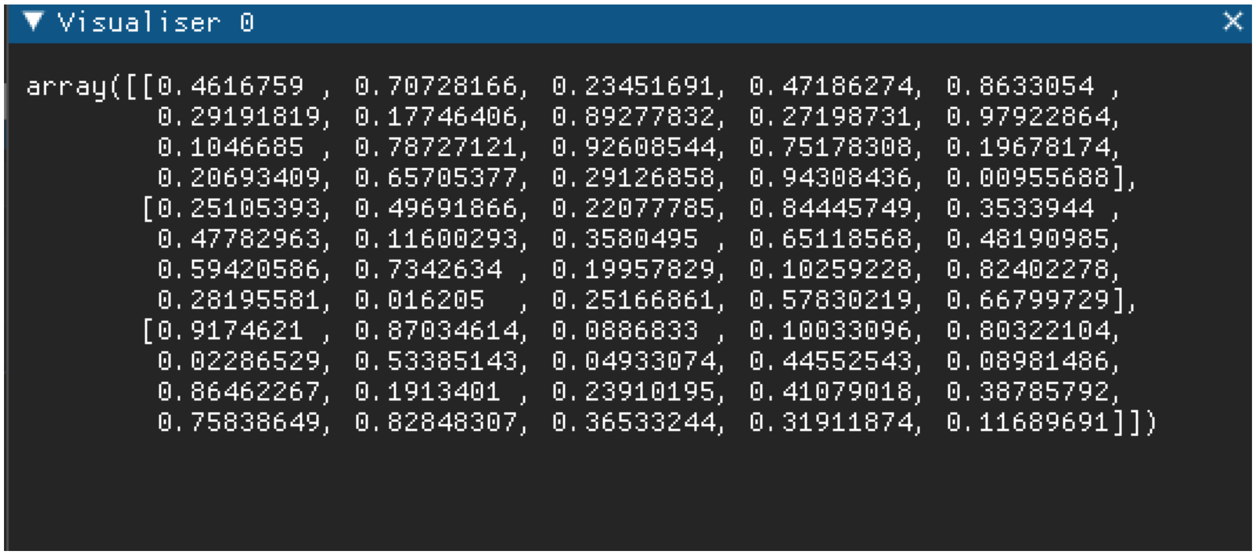
8.2 The visualisation elements

1. The Image



The VisualisationDPG class uses OpenCV's imshow to display images.

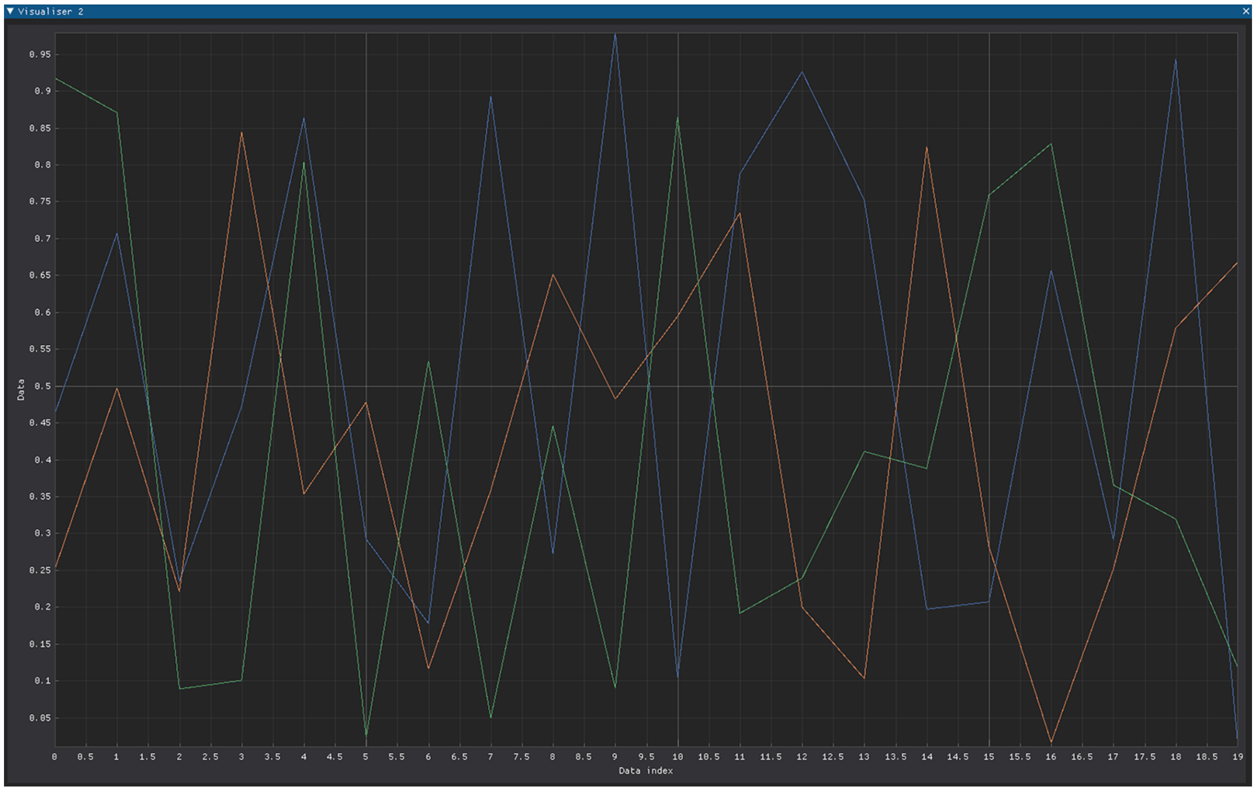
2. The Value



```
array([[0.4616759 , 0.70728166, 0.23451691, 0.47186274, 0.8633054 ,
         0.29191819, 0.17746406, 0.89277832, 0.27198731, 0.97922864,
         0.1046685 , 0.78727121, 0.92608544, 0.75178308, 0.19678174,
         0.20693409, 0.65705377, 0.29126858, 0.94308436, 0.00955688],
        [0.25105393, 0.49691866, 0.22077785, 0.84445749, 0.3533944 ,
         0.47782963, 0.11600293, 0.3580495 , 0.65118568, 0.48190985,
         0.59420586, 0.7342634 , 0.19957829, 0.10259228, 0.82402278,
         0.28195581, 0.016205  , 0.25166861, 0.57830219, 0.66799729],
        [0.9174621 , 0.87034614, 0.0886833 , 0.10033096, 0.80322104,
         0.02286529, 0.53385143, 0.04933074, 0.44552543, 0.08981486,
         0.86462267, 0.1913401 , 0.23910195, 0.41079018, 0.38785792,
         0.75838649, 0.82848307, 0.36533244, 0.31911874, 0.11689691]])
```

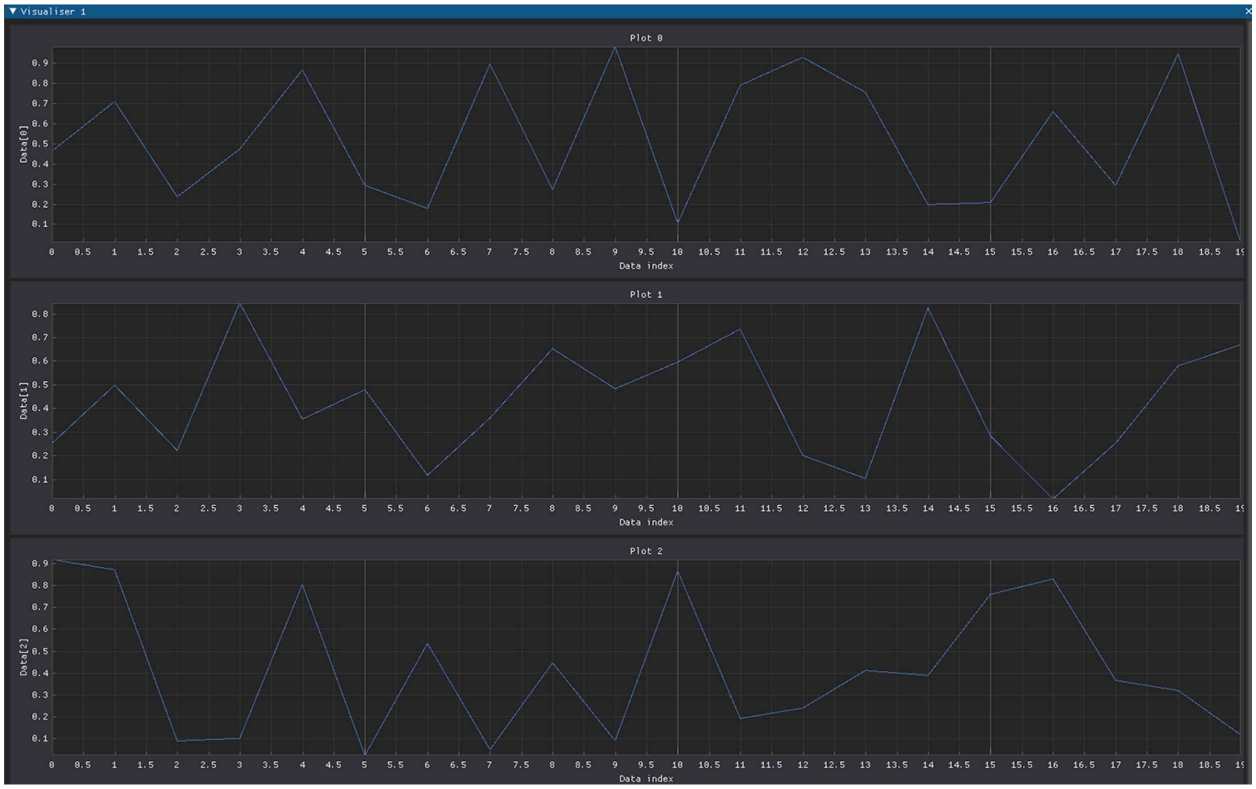
The VisualisationDPG object will open a text box where the values of the variables passed to the object for visualisation will be displayed. As mentioned above the buffer here means how many values from past iterations are kept visible in the text box.

3. The Single Pane Plot



The Single Pane Plot will display 1D or 2D numpy arrays in a single window. In the case of 2D arrays with shape (x, y), there will be x number of lines, each of y number of points.

4. The Multi Pane Plot



The Multi Pane Plot will display 2D numpy arrays with shape (x, y) over x plots each with y number of points.

CHAPTER

NINE

THE SAVING STATE SYSTEM

Heron allows a Node developer to quickly set up a system that saves the parameters, every time they update, and any data the worker script has access to at every iteration of the Node's worker function.

9.1 A Node's Saving System

If the Saving system is used Heron will generate up to two pandas Dataframes. One will store all the parameters, with columns names defined by the developer, and with every row corresponding to the parameters of the Node when any of them was changed by the user. This dataframe will be called 'Parameters.df'. The other dataframe (called Substate.df) has column names also defined by the Node's developer and saves any piece of information the developer assigns at each iteration of the worker function.

In both the Parameters and Substate dataframes the rows are also timestamped through the 'DateTime' column and there is a column (called 'WorkerIndex') that registers the current iteration of the worker function.

The Saving System needs two things to automatically start saving information. One is an implementation in the Node (see bellow how this is done). The other is a folder path in the 'Save the Node State to directory' text entry of the Node's secondary window (see *The Editor*). If any of these two things are missing then the system just doesn't run. That means that if there is a Node implementation then it is up to the user to put a folder path in to get the system to work (or not). On the other hand if the Node hasn't implemented the system then adding a folder path will do nothing (including not notifying the user that nothing will happen).

9.2 Implementation in the Node

Access to Heron's implementation of the Save State system is done through the worker object in the Node's initialisation function and the Source Nodes' worker function and through the savenodestate_update_substate_df function that is passed as an argument to the worker functions of the Transform and Sink Nodes every time they are called by Heron.

More specifically the worker_object has two methods (savenodestate_create_parameters_df and savenodestate_update_substate_df) and a variable (num_of_iters_to_update_savenodestate_substate) that pertain to the Save State system.

9.2.1 Saving the parameters

A developer who wishes to have a Node that is able to save all its parameters' changes through the lifetime of a running Graph can add the following line either to the initialisation function of any Node or to the pre infinite loop part of the worker function of Source Nodes.

```
worker_object.savenodestate_create_parameters_df(parameter_var_1=global_var_1, parameter_
↪var_2=global_var_2,
                                                 parameter_var_3=global_var_3, parameter_
↪var_4=global_var_4)
```

The parameter_var_1, parameter_var_2, etc. will become the column names of the saved dataframe. They can be any names the developer wants but using the same names as the parameters of the Node is considered good practice. The global_var_1, global_var_2 variables will become the initial values (1st row of the dataframe) and need to be valid variables in the Node at the time this method is called.

Given the above code and a Save State System folder path in the secondary window of the Node, the system will then save all parameter values any time the user changes any of the parameters at the Node's main GUI.

9.2.2 Saving part of the Node's state

Saving some of the Node's state in the Node's Save State System can be achieved with one of two different methods given the type of Node. Irrespective of the Node type though, for the implementation of the Substate saving system to operate the parameters saving implementation must be present (practically the savenodestate_create_parameters_df must be present before the following Substate implementation can work).

In the case of Source Nodes (which expose the full SourceWorker worker object in the worker function of the worker script) one can simply add (usually in the infinite loop part of the worker function) the following code:

```
worker_object.savenodestate_update_substate_df(some_name_1=some_data_1, some_name_2=some_
↪data_2)
```

The Transform and Sink Nodes on the other hand, have worker functions which do not expose the whole worker object, but expose the function savenodestate_update_substate_df as an argument. So if a Node developer wishes to use the Save State system in Transform and Sink Nodes they must define their worker functions with savenodestate_update_substate_df as a third argument (after the data and parameters ones) and then call that function exactly like above inside the worker function:

```
def worker_function(parameters, data, savenodestate_update_substate_df):
    ...
    savenodestate_update_substate_df(some_name_1=some_data_1, some_name_2=some_data_2)
    ...
```

Here, as in the case of the parameters, the 'some_name_1', etc. names of the arguments of the savenodestate_update_substate_df function are going to become the names of the dataframe's columns. The some_data_1, etc values are data that the user wishes to save.

9.2.3 Controlling the update of the dataframes

The Parameters dataframe, as mentioned above is automatically updates every time the user changes any of the parameters (which is not a very frequent action). The update of the Substate dataframe though to the hard disk can be an expensive operation especially if it needs to happen many times per second (at the speed at which a fast Node might need to call its worker function) and/or that data saved are large.

Currently the Node's developer and users can control when Heron will save the dataframe (which is constantly being updated in RAM) to disk. This is achieved either through a global variable found in the constants script called NUMBER_OF_ITTERATIONS_BEFORE_SAVENODESTATE_SUBSTATE_SAVE or through a Node specific variable called num_of_iters_to_update_savenodestate_substate the worker object exposes. If the num_of_iters_to_update_savenodestate_substate is set then it takes precedence over the global variable.

If the num_of_iters_to_update_savenodestate_substate (or the NUMBER_OF_ITTERATIONS_BEFORE_SAVENODESTATE_SUBST
when no num_of_iters_to_update_savenodestate_substate is set for the Node) is set to -1 then the Relic system will not update the Substate dataframe to disk until the process is about to terminate. There is a tradeoff here. If the Save State system's dataframe is saved to disk only as the process closes down then any crash that would abnormally terminate the process without allowing it to run its end_of_life function will mean loss of the Substate dataframe. On the other hand long running processes in machines with small RAM might run out of memory while keeping the dataframe in RAM.

9.2.4 Loading saved DataFrames

The Parameters.df and Substate.df are pandas dataframes that can be loaded later on with the command:

```python
import pandas as pd

substate_file = r'The Save State System directory/The Node Name/Substate.df'
parameters_file = r'The Save State System directory/The Node Name/Parameters.df'

substate_df = pd.read_pickle(substate_file)
parameters_df = pd.read_pickle(parameters_file)
```

CHAPTER

TEN

SYNCHRONISATION

Heron, as an event based system, running over multiple processes and machines, does not rely on a single clock to generate synchronised behaviour. Yet, as an experiment pipeline generator, it needs to allow users to fully time match the different events and behaviours spawned by the different Nodes. To do this, Heron uses a number of logging systems, which, when combined, offer the required time matching capacity. The individual logging systems have been described in the *Debugging* section for the logging system keeping information on the communication between Nodes and the *The Saving State System* section for the system keeping information on the internal activity of each Node.

In what follows we will describe how those two systems can be used in conjunction and together with some external (hardware based) logging for the case of multiple machines, can generate a full record of the timing of every event that a Heron pipeline generates.

10.1 Hardware

Within a single machine, Heron timestamps all logged events using the machine's clock. This of course breaks down when the pipeline includes Nodes running in multiple machines. The only way to time match events that are generated in different machines (at the required accuracy of an experimental setup) is to have an external data acquisition (DAQ) device that will register the timing of different events that happen on separate machines. Such a DAQ should save its data through a Heron Node running in the Heron GUI machine and have the saved data packets timestamped through Heron's logging system.

10.2 Software

A Node can use Heron's Node Save State System (see *The Saving State System*) to record information available to the worker process every time the worker function runs (for Transforms and Sinks) or goes around once in its infinite loop (for Sources). That allows the worker process to register the time and index of the packet it generates together with whatever other information it needs to save (the timestamping happens at the moment the savenodestate_update_substate_df function is called in the worker function).

The worker process of a Transform or a Sink though has no information about the packet that triggered the worker function, coming from a left connected Node. This information is saved in the Node's com process log file (described in *Debugging*). There, Heron saves both the timestamp and the index of the incoming packet from the left Node and the corresponding timestamp and the index of the packet the current worker function generates. In this way one can connect which packet from a left Node has triggered the production of which packet from a right Node.

10.3 Examples

To make the above more concrete lets give here two examples of simple pipelines and how one would go about time matching their outputs.

10.3.1 Within machine (saving a 120fps camera)

One of Heron's Nodes is a Spinnaker Camera. Some of the FLIR cameras this Node can capture can reach large frames per second (FPS) speeds, too high for a standard Save CV2 Video Node to be fast enough to save appropriately. That is why within the Vision repo there is a Save FFMPEG Video Node.

Let's assume we have a pipeline where a 120fps FLIR camera is captured by a Spinnaker Node and this feeds into a Save FFMPEG Video Node. Now, there are three possible places where frames can be lost:

1. In the capturing from the camera to the Spinnaker Node.
2. In the transfer from the Spinnaker to the FFMPEG Node.
3. In the save into the video done in the FFMPEG Node.

Let's also assume that our application does not require 100% perfect capture (i.e. zero dropped frames from camera to video file) but (as in most experimental cases) needs to allow the user to know exactly which frames have been dropped where and to be able to assign the real time a frame has been captured by the camera sensor to the corresponding video frame saved in the FFMPEG generated video file. They would also need to be able to assign every frame in the final video to any other event in the Heron pipline (e.g. a press of a button). In order to achieve this the user should:

1. Assign a save directory to the "Save the Node State to director:" entry of the secondary window of the Spinnaker Node.
2. Assign a log filename in the "Logfile or Verbosity Level" entry of the secondary window of the Save FFMPEG Video Node.

The Spinnaker Node saves the id and time of each captured frame as specified from the camera's hardware together with the corresponding index and timestamp of the Node's loop iteration. This way, any frames generated by the camera but are not captured by the Node become obvious.

The logfile generated by the FFMPEG Node's com process defines the index and timestamp this process assigns to every packet (frame) that reaches it from the Spinnaker Node. Here, any frame that doesn't make it across the Nodes will become obvious but more importantly this log file will generate a one to one correspondence between the indices of the frames that pass through as they are generated by the Spinnaker Node and the indices generated by the FFMPEG Node. Finally the FFMPEG Node uses an FFMPEG pipeline to write every receiving frame into an encoded video file. This needs to be optimised (in the pipeline the FFMPEG Node defines) so that the number of frames in the final video should be equal to the number of indices in the log file of the FFMPEG Node (that means the 3. above should never happen).

Following the above procedure one can create a one to one connection between any video frame and the time the computer captured that frame from the camera. They can also use the computer timestamp of any frame (the timestamp that marks when the frame was captured by the Spinnaker Node) to assign every video frame to any other timestamped event in Heron's pipeline as long as this happened in a Node running in the same Heron GUI machine.

10.3.2 Across Machines (time matching frames of different cameras)

Now let's assume that in the above described Heron pipeline there is also an Arducam camera Node running on a Raspberry Pi connected to the same LAN as the Heron GUI computer. The goal here is to be able to assign every frame from the FLIR camera saved onto the FFMPEG generated video to every frame captured by the Arducam camera and saved to the Arducam video. Here a user would need to again provide a save directory to the Arducam Node's secondary window. This way the Arducam Node would save the timestamp and the index of the captured frame, which, because the saving of the video happens within the same Arducam Node, would correspond to the equivalent saved frame in the video.

Yet that would not be enough to time match the frames of the two videos to each other, since the timestamps for the Arducam frames are generated by a different computer to the ones assigned to the FFMPEG Video. The only way around this would be for the user to have a DAQ that captures hardware triggers for both the FLIR camera and the Arducam one and save those through a separate Node (e.g. the NIDAX Node). At the same time the Arducam Node can capture a TTL pulse that comes into the Ras Pi's GPIO and save its index and time onto the Save State dataframe together with the index of the captured frame. This saved TTL pulse is the same trigger that triggers the Arducam camera and is also saved in the DAQ. This way the DAQ becomes the central clock onto which the camera captured frames from both camera systems get registered. Through this common clock and the use of Heron's Save State dataframes and com process log files, all events on both computers can be time matched to any other event.

CHAPTER

ELEVEN

DEBUGGING

Things go wrong, so read on.

11.1 Usage and limitations of the python debugging tools

By far the most common way to debug the construction of a Node (i.e. the creation of the xxx_worker.py script) and its subsequent use in a pipeline is to use the Python standard debugging tools. That is the Python debugger and the print method. Both those will work just fine as long as the Node's worker process runs on the same machine that Heron's GUI is also running. The print method will print to the Heron's command line window while a debugger will stop the xxx_worker.py script in exactly the same way as if it was called by Python (which it is, just through the com process).

11.2 The logging system

If the worker process is running on a different machine then neither the debugger will see it nor any print statements will pass their output to the main machine and Heron's command line window. In order to solve this (and to also keep a general eye on Heron's overall performance) Heron has implemented two levels of Python logging.

11.2.1 The com process's verbosity and log

Each Node's secondary window provides an entry named 'Log file or Verbosity level'. There a user can set either a number (for now all numbers do the same thing but in the future there might be more verbosity levels) or a full path to a log file.

If there is a number then the com process of the Node will output to Heron's command line window a detailed description of every message as it leaves the com process. This includes the different times it took for the message to pass through the different processes. For example a Transform com process will output the time it took the message to get from the com process of the upstream Node to the current com process, the time it took to be send to the worker process, be transformed and arrive back at the com process and the time it took to be sent to the downstream Node's com process.

If the 'Log file or Verbosity level' entry gets a path to a log file (xxx/xxx/some_file.log) then the com process will output some basic info to that file every time it processes a message. This info will involve the index of the message in the current Node (i.e. how many messages this Node has processed), the time the com process finishes processing the message and if the message is coming from an upstream Node also the index of the message in that Node and the topic of the message (i.e. the string that tells which Node and which output the message came from and which input the message has arrived into this Node). .. note:

```
These log files are very useful to be able to track messages across Nodes. How they can␣
↪be used to time match packets
across processes is further described in the :doc:`synchronisation` section.
```

11.2.2 The global and local log files

Heron offers two other logging systems so that Node developers can log information that is happening on the worker processes of the Nodes they are developing.

The first is the global logging system. By importing Python's logging package a Node developer can use it's functions, e.g.:

```
logging.debug('message)
logging.error('message)
```

in order to log messages to a Heron.log file that gets automatically generated in the Heron/Heron directory. That file will also log messages from any package that uses logging that is being used in Heron's functionality (for example one will see a lot of messages there from paramiko as Node sockets connect to the forwarders' processes sockets).

Another important use of the global Heron/Heron/Heron.log file is that it registers any traces produced when the code breaks and a Node in the pipeline shuts down. So if you see a message in Heron's command line about a Node killing itself (which is the output of the end of life functionality) then check in the Heron.log file for a trace as to what exactly happened.

Heron automatically produces this global log file on any machine that runs Heron during a pipeline. So a developer can register debugging messages in a worker script of a Node that runs on a different machine to Heron's GUI. On that machine Heron will generate a Heron.log file and the messages will be registered there. The above point about abnormal termination and stack trace registering applies also to worker processes running on separate machines, so if something goes wrong one should first check the Heron.log file on the machine that the worker process of the Node is running.

Heron also offers a local way to log information. In the general_utilities there is a function called setup_logger. Using this will generate a separate logger with a specified log file that the worker script of a Node can use to log information. Use this as follows:

```
from Heron import general_utils as gu
logger = gu.setup_logger('Name of Logger', full_path_log_file_name)

logger.info('Some information')
logger.debug('Maybe we should look into that')
logger.error('Aarghh, something is wrong')
```

Of course if the worker script is meant to run on a different machine the full_path_log_file_name must make sense for the machine that it runs on since the logger is not designed to pass its messages to different machines.

11.3 Hanging processes

When Heron crashes all processes should receive a kill command and stop themselves after HEARTBEAT_RATE * HEARTBEATS_TO_DEATH seconds (these variables are defined in the constants.py file). The same goes for a process that crashed because there was a bug in its code. If the error itself doesn't kill the process but makes it unresponsive then it will kill itself after the above specified amount of time.

Yet, sometimes, some process do not kill themselves. This happens very rarely but it is possible. When that happens, restarting the Graph (even from a new Heron GUI) will throw an error claiming that the sockets required for communication are not available. The only way to stop such a process is to access it and kill it manually in whatever the OS provides. For example in Windows this can be done from TaskManager while from Linux with the top command.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search