This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

Joule: A Real Time Framework for Decentralized Sensor Networks

John Donnal
Weapons and Systems Engineering
United States Naval Academy
Annapolis, Maryland 21402
Email: donnal@usna.edu

***Abstract*—Traditional IoT sensor nodes transmit raw data to centralized servers for analysis and storage. At higher data rates this architecture is no longer feasible. Instead data must be processed at the lowest level possible to include the sensor node itself. The Joule framework enables decentralized data processing by distributing computation into modules which are dynamically linked at run time by a process supervisor. Modules may be collocated on a single machine or distributed across a network. Data is stored where it is processed creating a fully decentralized architecture. Processed data can be queried by centralized servers or delivered directly to the end user. This allows high bandwidth sensors to be incorporated seamlessly into existing IoT deployments without depleting network bandwidth or server resources. This paper presents the Joule framework as well as performance benchmarks and deployment case studies.**

I. Introduction

Traditional Internet of Things (IoT) frameworks require sensors to transmit raw data to a central server for analysis and storage. High bandwidth sensors with unreliable network connectivity are difficult to integrate into these frameworks. There is a pressing need to shift computation away from the data center and out to the network edge. Single board computers (SBC's) such as the Raspberry Pi offer an ideal platform for edge computing. The SBC market evolved primarily from the abundance of high performance low energy chip sets designed for mobile phones. These platforms have robust hardware support for multi-threading, floating point, and network connectivity that are not present in microcontrollers, yet they consume single digit watts making it possible to deploy them in embedded environments where desktop and server processors cannot operate due to power and heat dissipation requirements.

Decentralization reduces the server load and the burden on communication networks, but places significant demands on the client. In order to process and store data locally, each client requires an assortment of low level hardware drivers and data management routines as well as high level signal processing and machine learning algorithms. These are generally implemented as proprietary software stacks that vary from platform to platform. Maintaining such a complex code base for a wide variety of clients is expensive and error-prone.

Joule provides a common framework for implementing decentralized data processing eliminating the need for costly customized software stacks. It is optimized for SBC's but can run on any Linux platform. Joule distributes computation into independent executable modules. Modules are loosely coupled by streams and can be updated or replaced independently. This provides flexibility to meet rapidly changing requirements common in IoT environments. With Joule, high bandwidth sensors can operate autonomously or coexist with centralized infrastructure by transmitting fully processed lower bandwidth data to upstream servers.

A. Real Time Processing Frameworks

The large influx of data both from physical sensors as well as software analytics has driven the development of a variety of real time processing frameworks. Apache Storm [1] is a general purpose stream processor that enjoys wide industry adoption. Recent work by [2], [3] improves cluster flexibility with dynamic scheduling and resource balancing. Apache Kafka [4] is a similar framework that provides real time messaging and is often used in conjunction with Storm [5]. Several other architectures have been developed for particular use cases such as [6] for cyclic streams and [7] for elastic computation. These frameworks assume data is processed by well connected servers. That is, the data is still centralized. Joule executes directly on the client node providing completely decentralized operation without any dependence on a network connection. Furthermore Joule, unlike the general purpose Kafka and Storm frameworks, is designed specifically for continuous time series. This simplifies the software abstractions and reduces system overhead which is critical in a resource constrained embedded environment.

B. Design Goals

Joule is designed to meet three primary objectives: speed, flexibility, and security. First and foremost Joule provides an efficient architecture to handle the acquisition and processing of high bandwidth data. Speed by itself is not difficult to achieve, but maintaining flexibility at the same time is a challenging problem. Embedded IoT platforms usually run monolithic binaries or a thin real time operating system (RTOS). These architectures provide full access to the hardware and therefore run at optimal speeds, but this is at the cost of flexibility as updates generally require a firmware flash. Monolithic binaries are also difficult to maintain because dependencies are not well defined. A seemingly insignificant

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

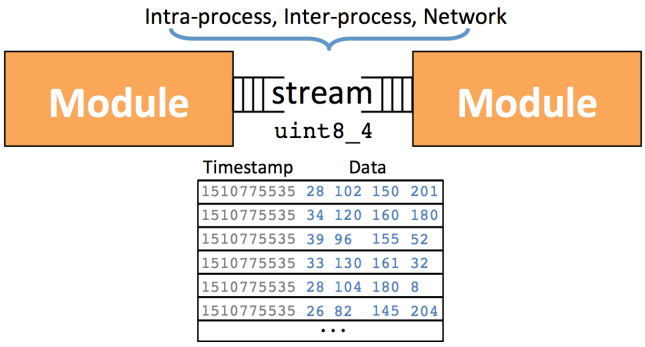

Fig. 1. Joule is a modular processing framework. Streams are strongly typed data flows that connect modules. Streams can connect modules within a single Linux process, between processes, or over a network connection

change can have ripple affects across the code base, some of which may not be detected until the code has been deployed.

Joule uses composable modules that provide flexibility while still maintaining a high throughput computational architecture. Modules are connected by strongly typed data flows called streams as shown in Fig. 1. Modules do not need any knowledge of where or how their inputs are produced and do not need any knowledge of where or how their outputs are consumed. This makes modules easier to develop than monolithic binaries because programmers can work on code isolated to their domain of expertise. They are easier to test because their inputs and outputs (dependencies) are well defined, and they are easier to maintain because modules can be upgraded and replaced independently.

The final design goal of Joule is security. The system should be resilient both to malicious actors as well as unintentional errors in user code. In this sense, security is agnostic to intent. If a poorly written routine consumes excessive processor resources it becomes an (unintentional) denial of service attack. Joule protects the system from malfunctioning modules by running each module as a separate process. The Linux kernel provides robust tools to manage process security and isolation. Using cgroups, the kernel can restrict the CPU and memory usage of a process to predefined limits or isolate a process in a virtual root (chroot) environment with limited or no access to the underlying file system [8]. Joule monitors process execution, automatically restarting any modules that fail. As the module restarts, Joule buffers incoming data until the process is ready to receive it, and any subscribers to the module outputs continue operating without interruption.

The layout of the paper is as follows: Section II presents the Joule framework and theory of operation. Section III introduces an application programming interface (API) for designing Joule modules. Section IV provides benchmarks on common IoT platforms. Finally, Section V presents case study results.

II. Framework Implementation

The Joule Framework consists of a process supervisor, jouled, and one or more modules which perform the data processing. Modules are connected by strongly typed timestamped data flows called streams. Modules may have multiple inputs and outputs and streams may branch to connect multiple modules. This enables complex pipeline designs that span multiple processes and machines.

A. Modules

Modules are configured using text files. The file is divided into `[sections]` of `key=value` attributes. An example module configuration is shown in Listing 1.

```
[Main]
#required
name = RMS Filter
exec_cmd = /path/to/executable --args
#optional
description = Compute 2-D RMS

[Inputs]
x = /surface/accel/x
y = /surface/accel/y
# additional inputs...

[Outputs]
rms = /surface/accel/rms
# additional outputs...
```

Listing 1. Example module configuration file

The [Main] section contains general attributes. The name must be unique and is used to identify modules in the Joule command line interface (CLI). The exec_cmd is the absolute path to the executable with any necessary command line arguments. Modules may be any executable script or compiled binary. The [Inputs] and [Outputs] sections contain stream attributes. In this example the `/surface/accel/x` stream is connected to the module's `x` input. Outputs are data sources and inputs are data sinks. A stream must have a single source and may have multiple sinks. This requirement means output bindings are unique while input bindings may be shared by multiple modules.

B. Streams

Streams are timestamped data arrays. Timestamps are in UNIX Time (microseconds since January 1 1970). Microsecond resolution supports sensors with bandwidths up to 500KHz. The data values are strongly typed which enables efficient binary transmission between modules. Like modules, streams are configured with text files. An example configuration is shown in Listing 2.

The [Main] section contains general attributes. The path attribute is a unique identifier which provides a logical hierarchy for grouping streams into folders. This is strictly a logical representation and does not affect how streams are processed or persisted to disk. The data type is the binary data representation and bit width of the elements. All elements share the same type. Types may be `float`, `int`, or `uint` of `1`, `2`, `4`, or `8` bytes. Streams are referred to by their composite type which is the concatenation of the element type and the element count. The stream in Listing 2 has two four byte (32

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

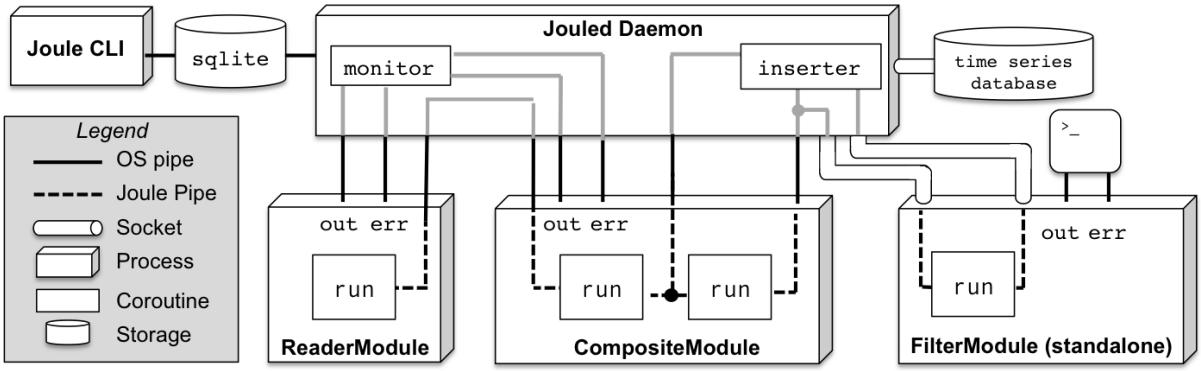

Fig. 2. The Joule Framework. The modular data processing pipeline is controlled by `jouled`. Modules are connected by data streams. Streams use a variety of transport mechanisms including intra-process queues, OS pipes, and network sockets. Persisted streams are stored in a local time series database and runtime metadata including module logs is cached in a relational database (SQLite). The Client API provides a common abstraction for data streams called Joule Pipes, as well as three fundamental module types: `ReaderModule`, `FilterModule`, and `CompositeModule`.

bit) floats so it has a composite type of `float32_2`. The stream in Fig. 1 has four eight bit unsigned integer elements so the composite type is `uint8_4`. The timestamp is always an `int64` regardless of the element type. Storing time in a fixed width type avoids floating point precision errors.

```
[Main]
path = /surface/accel/rms
datatype = float32
keep = 1w
decimate = True

[Element1]
name = magnitude
plottable = yes
offset = 0.0
scale_factor = 1.0

[Element2]
name = angle
plottable = yes
offset = 0.0
scale_factor = 57.3 #convert to degrees
```

Listing 2. Example stream configuration file

The keep attribute is a time duration in units of years (`y`), months (`m`), weeks (`w`) or days (`d`) that indicates how long data is stored. A keep value of `false` means that the stream is ephemeral and will not be persisted to disk. During module development it is helpful to store some amount of data for all streams, while in production reducing the number of persisted streams improves speed and reduces disk usage. The decimate attribute is a boolean value which indicates whether a persisted stream should be stored as a decimated hierarchy. Decimated streams can be quickly visualized at arbitrary time scales as described in [9]. Decimation roughly doubles the amount of storage required for a stream.

The `[ElementX]` sections describe the data values. Each element must have a unique name. The additional attributes provide optional metadata useful for visualization frameworks such as the interface described in [10].

C. Supervisor Process (`jouled`)

The framework is managed by the `jouled` daemon. This runs as a service and is started during the system boot process. It spawns modules, tracks their state, and restarts any ones that fail. Additionally, it routes data between modules, stores persisted data to a time series database described in [9], and records module meta data including CPU consumption, memory usage, and logs to a relational database that can be queried by the Joule CLI.

The process is fundamentally input/output (I/O) bound by the module data rates. This must be the case because if jouled was computationally bound there would be no CPU time available to the modules. In an I/O bound process there is some blocking request to read or write data that stalls execution. In jouled blocking I/O occurs in module data routing, network access, and disk access. These actions must be handled concurrently so it is not practical to pause execution waiting for an I/O request to complete.

Instead, jouled uses an asynchronous execution model. Asynchronous programs use coroutines that emulate concurrent execution in a single thread. Figure 3 shows how an asynchronous program and a multi-threaded program execute an identical computational task. When a coroutine is executing it is the only code running which eliminates the need for thread safe data structures and mutexes. When a coroutine executes an I/O request it yields execution back to the event loop which schedules the next available coroutine. The blocked coroutine is marked as available when its I/O request has completed. Coroutines may also explicitly yield execution to another coroutine or simply request to be rescheduled after a set time (ie sleep). In addition to elegantly handling blocking I/O and simplifying data management, coroutines are more efficient at context switching than threads. Threads are only advantageous for computationally bound programs when the hardware can support their simultaneous execution.

The supervisor begins by parsing the configuration files described in Sections II-A and II-B to construct an implicit

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

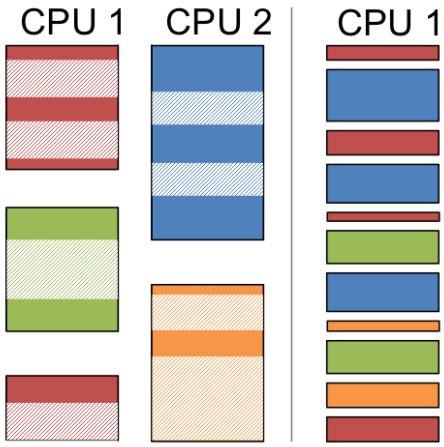

Fig. 3. Multi-threading (left) vs asynchronous programming (right) for an I/O bound process. Colors represent threads (left) and coroutines (right). Hashed colors indicate execution stalls for blocking I/O, and white space indicates context switching overhead. The asynchronous implementation consumes half as many CPU resources to execute the same programming task.

directed acyclic graph (DAG) of module and stream dependencies. The boot-up sequence is shown in Listing 3. The supervisor starts modules with empty inputs first then iteratively starts any module with input streams that are produced by currently executing modules. If the supervisor iterates through the module list and is unable to start any module it terminates the boot-up sequence. Modules cannot be started either because they form a cyclic dependency graph or because there is no module producing their inputs.

```
streams = read_stream_configs
modules = read_module_configs

#iteratively start valid modules
worked_paths = []
while(state_changed):
  state_changed = false
  for module in modules:
    if(module.inputs not in streams or
       module.outputs not in streams):
       continue #skip invalid module

   if module.outputs exist in worked_paths
      execute(module)
      worked_paths.append(module.outputs)
      state_changed = true
```

Listing 3. Pseudocode for the supervisor boot-up sequence

D. Local Modules

Figure 2 illustrates how jouled manages the module processing pipeline. For locally executing modules, jouled injects streams through inter-process communication (IPC) pipes. A separate pipe is used for each input and output stream. Prior to calling fork/exec to spawn the module, the supervisor-facing pipe ends are marked `FD_CLOEXEC` which means they are closed after the exec and not available to the module process. This ensures modules respect the one way flow of data from inputs to outputs. IPC pipes are also used to redirect the module's `stdout` and `stderr` to a supervisor coroutine which persists their output to a relational database. This database can be queried with the command line interface to track module execution and debug problems in the processing pipeline.

The reassignment of `stdout`, and `stderr` is transparent to the module, but the additional file descriptors used for stream connections are explicitly communicated to the module via command line arguments. This is done by appending a JavaScript Object Notation (JSON) encoded dictionary of (name, descriptor) pairs to the module's `exec_cmd`. For example the module configured in Section II-A would have two read file descriptors and one write file descriptor and would be executed with the additional argument string:

```
--joule_pipes = "{
    inputs: { x: 5, y: 7 },
    outputs: {rms: 8}
  }"
```

The particular values of these descriptors is arbitrary and depends on the order in which the modules are spawned by the supervisor.

E. Remote Modules

Remote modules connect to jouled through network sockets. A server coroutine provides a JSON application programming interface (API) with end points for connecting remote inputs to local streams and connecting remote outputs to the local time series database. Clients request stream inputs by subscribing to the stream path. The stream must be produced by a currently executing module. Once subscribed, jouled spawns a coroutine to transmit stream data over the socket as it is produced by the local module. To inject a remote stream into the local time series database, the client provides the full stream configuration as a JSON object. jouled first verifies this stream is not currently being produced by a local module. It then checks if this path exists in the local database, if it does the existing stream data type must match the requested data type. Once these checks are completed jouled spawns a coroutine to receive stream data over the socket which it then stores in the local database.

III. Client API

The Client Application Programming Interface (API) simplifies module development by providing a high level interface to data streams as well as base classes for common module types. Modules are strictly I/O bound. This must be the case because a computationally bound module cannot support continuous streaming data. Therefore, like jouled, modules work well with an asynchronous programming model. Streams are the core I/O component. These are provided to a module over various transport protocols. The Client API provides a common asynchronous abstraction for managing data streams called Joule Pipes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

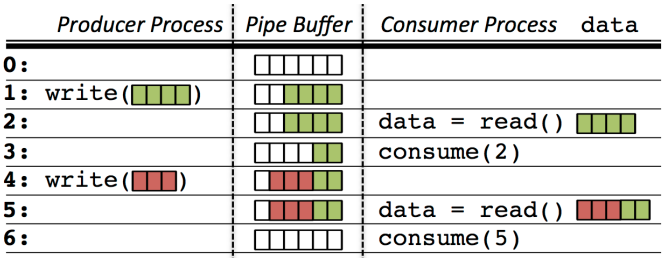

Fig. 4. Joule Pipes provided a high level interface to streams that is independent of the underlying transport protocol. Modules read and write to Joule Pipes using asynchronous coroutines. Explicitly separating the read and consume actions simplifies common streaming algorithms.

A. Joule Pipes

Joule Pipes provide a protocol independent interface to data streams. This decouples module design from pipeline implementation. The same module can run as a remote instance, local process, or composite coroutine without modification. Joule Pipes use a pair of queues to buffer incoming and outgoing data. The queues align data assembled from binary transport protocols into structured Numpy arrays [11]. Figure 4 illustrates how Joule Pipes move stream data between modules.

Output pipes have a single method, `write`, which transmits data through jouled to any modules that request the stream as an input. The data timestamps must be monotonically increasing and not overlap with any data already sent to jouled or present in the database. Input pipes have two methods: `read` and `consume`. `read` returns the current contents of the pipe buffer. The data remains in the buffer until explicitly removed by `consume`. This allows modules to manage how data is chunked, simplifying streaming algorithms that require a region of samples to compute an output value.

Joule Pipes decouple the transport transmission rates from the module's processing rate. When designing modules care must be taken to ensure that the code executes fast enough to handle streaming data. If a module's memory usage increases over time this indicates the module cannot keep up with its inputs and the Joule Pipe buffers are accumulating data.

B. Reader Modules

The Client API provides base classes for common module configurations. The ReaderModule class is designed for modules that read sensor data into the Joule Framework. An example implementation is shown in Listing 4.

Readers have no inputs and a single output as shown in Fig. 2. The `start` method creates a Joule Pipe for the output stream and then invokes the `run` coroutine which must be implemented by the child. This coroutine has two parameters, the command line arguments (parsed_args) and the output stream (output). Command line arguments are configured by implementing the `custom_args` method which provides a copy of the module's ArgumentParser instance. `run` should execute indefinitely. At a minimum it yields execution back to the event loop when writing to the output stream. Data is retrieved from a sensor (details not shown) and timestamped using the `time_now` utility function. The Joule Pipe `write` method requires a 2D array of timestamped data. For low bandwidth streams such as this example, samples may be inserted individually by promoting a single row to a 2D matrix. Higher bandwidth streams should use interpolated timestamps rather than repeated calls to `time_now` and batch writes with multiple rows of data.

```
1  from joule import ReaderModule, time_now
2  import numpy as np
3
4  class SensorReader(ReaderModule):
5
6    async def run(self, parsed_args,
7                  output):
8      while(1):
9        value = #read from sensor
10       data = [time_now(), value]
11       #output timestamped data
12       await output.write(np.array([data]))
13
14 if __name__ == "__main__":
15   r = SensorReader()
16   r.start()
```

Listing 4. A Joule Reader Module

ReaderModules are designed to be invoked by jouled, but they can also run as a standalone processes. A ReaderModule determines its execution environment by the presence of the `--joule_pipes` argument (see Section II-D). If this argument is missing, the module converts to standalone operation and connects the JoulePipe output to stdout redirecting the stream to the terminal. Standalone execution simplifies unit testing and allows ReaderModules to easily integrate with other streaming frameworks.

C. Filter Modules

The FilterModule class is designed for modules that process input streams into new outputs. The module shown Listing 5 is a streaming implementation of the median filter routine provided by the SciPy signal package [12].

The `start` method creates a JoulePipe for each input and output stream and then invokes the `run` coroutine which must be implemented by the child. `run` receives the parsed command line arguments and a dictionary of inputs and outputs indexed by the stream name specified in the module configuration file (see Section II-A). Like the ReaderModule, command line arguments may be configured by implementing the `custom_args` method.

Lines 12 and 13 retrieve the Joule Pipes for the streams associated with this module. More complex filters may have multiple inputs or outputs. The filter begins by reading data from the input stream into a structured Numpy array divided into timestamps and data. The data is filtered in place using the Scipy medfilt function. Filtering algorithms such as median require data before and after a sample to compute the output. In a streaming environment data is processed in blocks. Failing to account for this discretization produces artifacts at block boundaries where there is insufficient data to compute the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

output. In this median filter the first and last EDGE samples of the block are invalid so they are omitted from the output in Line 21. The call to consume on Line 23 leaves the last $2 \times$ EDGE samples in the pipe buffer to compensate for the invalid data regions. This execution sequence produces exactly the same result as a median filter run over the entire dataset at once.

```
1  from joule import FilterModule
2  from scipy.signal import medfilt
3
4  WINDOW = 21
5  EDGE = (WINDOW-1)/2
6
7  class MedianFilter(FilterModule):
8
9    async def run(self, parsed_args,
10                 inputs, outputs):
11     #retrieve joule pipes
12     input = inputs["raw"]
13     output = outputs["filtered"]
14     while(1):
15       #read new data
16       vals= await input.read()
17       #execute median filter in place
18       data = vals["data"]
19       data = np.medfilt(data,WINDOW)
20       #write out valid samples
21       await output.write(vals[EDGE:-EDGE,:])
22       #prepend trailing samples to next read
23       input.consume(len(vals)-2*EDGE)
24
25
26 if __name__ == "__main__":
27     r = MedianFilter()
28     r.start()
```

Listing 5. A Joule Filter Module

FilterModules are designed to be invoked by jouled but they can also run as standalone processes as shown in Fig. 2. A FilterModule determines its execution environment by the presence of the --joule_pipes argument. When executing as a standalone process the module and stream configuration files must be passed as additional command line arguments. The FilterModule uses these configuration files to request the appropriate network streams from the local jouled daemon. For live execution, all source streams must be currently produced by other modules. The FilterModule also supports historic execution which is useful during module development. In historic execution the FilterModule requests a specific interval of data rather than a live stream. Jouled provides historic data by connecting the module's input streams directly to the time series database instead of the output from another module. Historic execution is triggered by specifying starting and ending timestamps as additional command line arguments when running the module as a standalone process.

D. Composite Modules

Composite modules combine multiple modules into a single process as shown in Fig. 2. This improves data throughput by eliminating the overhead of process context switching and IPC. The module in Listing 6 combines the SensorReader and MedianFilter described previously.

```
1  from joule import CompositeModule,
2                    LocalPipe
3  from . import SensorReader, MedianFilter
4
5  class MedianSensor(CompositeModule):
6
7    async def setup(self, parsed_args,
8                    inputs, outputs):
9      #create child modules
10     reader = SensorReader()
11     filter = MedianFilter()
12     pipe = LocalPipe()
13     #connect interior streams
14     task1 = reader.run(parsed_args, pipe)
15     task2 = filter.run(parsed_args,
16                        {"raw": pipe},
17                         outputs)
18      #schedule modules for execution
19      return [task1, task2]
20
21 if __name__ == "__main__":
22   r = MedianSensor()
23   r.start()
```

Listing 6. A Joule Composite Module

The setup coroutine receives the parsed command line arguments and a dictionary of Joule Pipes associated with the module. These are the same parameters as FilterModule::run discussed in the previous section. Command line arguments can be configured by overriding the custom_args function. The nested modules are initialized and connected by a LocalPipe. LocalPipes are a Joule Pipe subclass that provide intra-process stream connections. LocalPipes also provide synchronous read and write methods that simplify unit testing. Calling run for each module returns a coroutine object. These are collected and returned for execution in the main event loop.

IV. Framework Performance

The Joule Framework is designed for multicore single board computers (SBC's) but it can be installed on any Linux distribution with systemd and Python3 (3.5 or higher). This section presents benchmark results from five platforms that illustrate the framework performance in a variety of hardware environments. The benchmarked systems are shown in Table I.

These platforms cover three distinct hardware environments. The BeagleBone and RaspberryPi are low cost, low power SBC's. The Intel NUC offers significantly more computational power with a slightly larger footprint and higher cost. Amazon Web Services (AWS) instances can be used for module development and anchor distributed Joule pipelines into traditional IoT infrastructure.

These benchmarks measure the processing requirements of jouled. This represents the overhead of the Joule framework over a monolithic binary implementation of a similar signal processing pipeline. jouled is a single threaded process which means it can only use one core. This is intentional as it

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

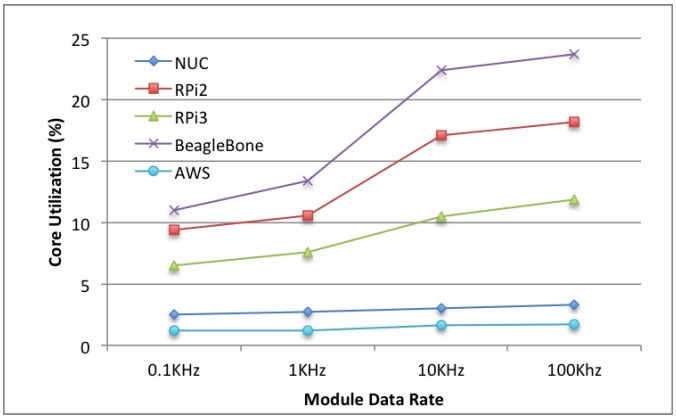

Fig. 5. Jouled is a single threaded application. This plot shows the workload for the core running jouled as a function of module data rate.

TABLE I
BENCHMARK HARDWARE PLATFORMS

Platform	CPU	Cores	Speed	RAM
BeagleBone	Cortex A8	1	1 GHz	0.5 GB
RaspberryPi 2	Cortex A7	4	0.9 GHz	1 GB
RaspberryPi 3	Cortex A53	4	1.2 GHz	1 GB
Intel NUC	Core i5	4	2.9 GHz	16 GB
AWS t2.nano	Xeon E5	1	2.4 GHz	0.5 GB

maximizes the amount of computational resources available for module execution. Figure 5 plots core utilization as a function of module data rate. In this benchmark, a ReaderModule produces a six element floating point (`float32_6`) stream that is decimated and stored locally.

The load percentages in Fig. 5 are only for a single core. Figure 6 plots the same data against the total CPU capacity. This is a more accurate representation of system load. The BeagleBone has a single core CPU so the core load in Fig. 5 is equivalent to the system load. The absence of multiple cores significantly decreases the BeagleBone's performance for this application. The RaspberryPi is available in a similar form factor at the same price so there is little reason to use the BagleBone as a Joule node. For this reason it is omitted from this and subsequent benchmark figures. For the other platforms, total overhead is less than 5% for all measured data rates.

As the number of modules increases there is a linear increase in CPU usage. Figure 7 plots jouled performance as a function of pipeline depth. In this benchmark a ReaderModule produces a `float32_6` stream at 1KHz which is then fed through a chain of FilterModules. The output of the final filter is decimated and stored. The total overhead is below 6% for all hardware platforms.

V. NON-INTRUSIVE LOAD MONITORING CASE STUDY

Non-Intrusive Load Monitors (NILM's) use high bandwidth current and voltage sensors to identify individual electrical loads from aggregate power waveforms [13]. NILM's can improve energy conservation, reduce maintenance costs, or detect anomalous activity [14], [15]. Unfortunately, NILM's have

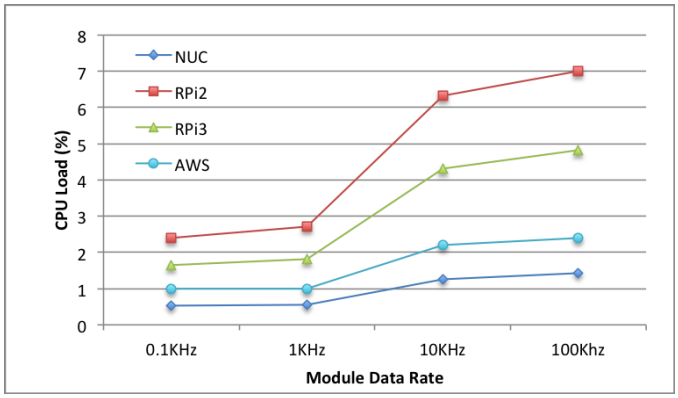

Fig. 6. This shows the same benchmark as Fig. 5 adjusted to reflect the total CPU capacity of the hardware platform. The BeagleBone is omitted because it has a single core processor.

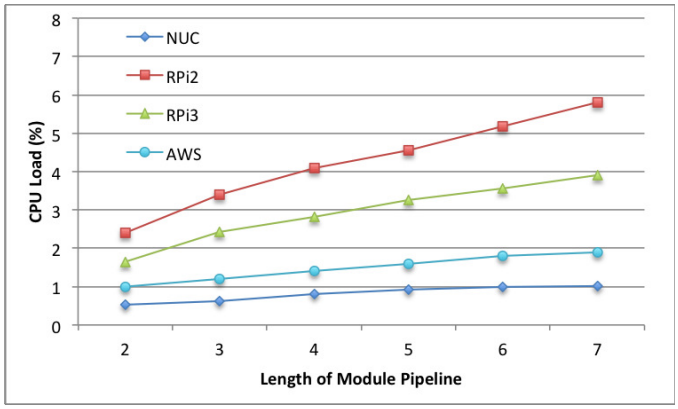

Fig. 7. CPU load versus pipeline depth. A single 1KHz `float32_6` stream is processed by a series of filter modules. This overhead can be reduced by combining modules into composites as described in Section III-D.

seen limited adoption in part because their high bandwidth data streams make them difficult to integrate with traditional IoT infrastructure. Using the Joule Framework, NILM's can perform the high bandwidth data processing locally. This allows them to operate as standalone devices or transmit lower bandwidth processed data to centralized servers for analysis.

The Yard Patrol fleet is a group of training vessels stationed at the US Naval Academy. These ships are designed for training rather than combat so they do not have the sophisticated sensor arrays that are standard on modern naval vessels. Installing NILM's on these ships improves their situational awareness by monitoring critical electrical loads. With the Joule Framework, the crew can receive real time feedback while the ship is underway, and when the ship is in port the data can be transferred to centralized servers for higher level analysis and comparison against other ships in the fleet.

Figure 8 shows a NILM installed in the engine room of the YP692. The NILM uses an array of electromagnetic sensors located around the outside of the power cable to measure three phase current and voltage. The sensors are sampled at 3kHz by an Atmel SAM4S microcontroller and transmitted over USB to a RaspberryPi 3 running the Joule Framework. Modules

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

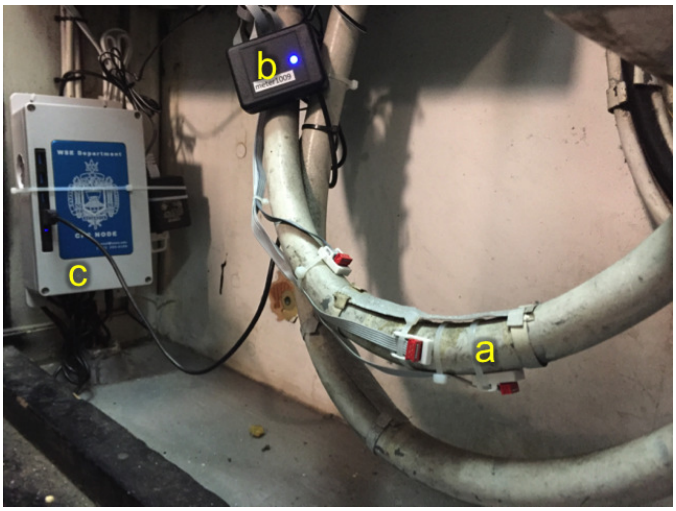

Fig. 8. Non-Intrusive Load Monitor installed on the YP692. Electromagnetic sensors (a) are placed radially around the primary power line. An ARM microcontroller (b) samples the sensors at 3KHz and transmits the data by USB to a Joule Node (c) for analysis and storage. After calibration this system can measure the three phase power consumption of the entire ship.

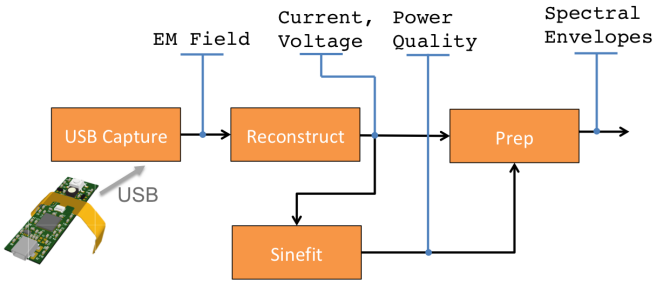

Fig. 9. Module configuration for the non-intrusive load monitor. The USB connected sensor produces eight 16-bit samples at 3KHz. The power spectral envelopes are decimated and stored locally. Total CPU load on a RaspberryPi 3 is less than 25%

then convert the raw sensor data into line frequency (60Hz) spectral envelopes. This represents a significant reduction in bandwidth. The spectral envelope data can then be processed by machine learning algorithms to identify individual loads (eg [16]–[18]).

Figure 9 shows the module architecture for computing power spectral envelopes from sensor data. The *USB Capture* module manages the sensor hardware and produces an eight element stream of electromagnetic (EM) field measurements at 3KHz. The *Reconstruct* module converts the field measurements into voltages and currents. The *Sinefit* module calculates multiple power quality metrics including line frequency. Finally, the *Prep* module uses the line frequency, current, and voltage streams to compute power spectral envelopes. The prep and sinefit algorithms are presented in [19]. The entire signal processing operation (jouled and associated modules) consumes only 23% of the RaspberryPi 3's CPU resources.

Figure 10 shows the fundamental spectral envelope (real power) as measured by the NILM during a typical week day.

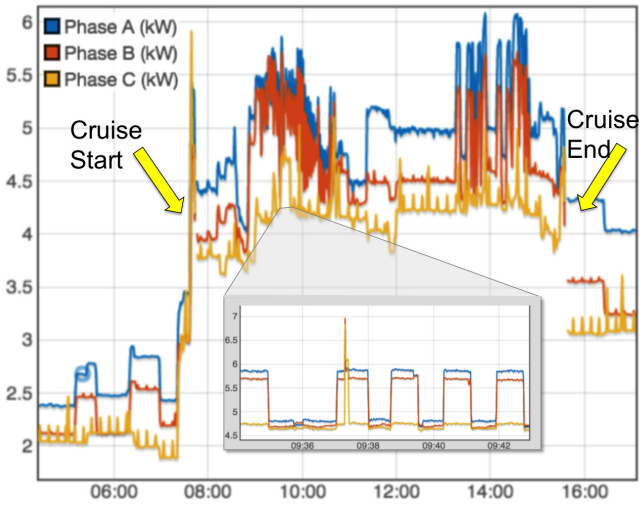

Fig. 10. Three phase power consumption of the YP692 sampled by a non-intrusive load monitor at 3KHz. The power interruptions indicate the ship is departing and then returning from a cruise. The higher baseline power after the cruise indicates the crew has left equipment running on the vessel.

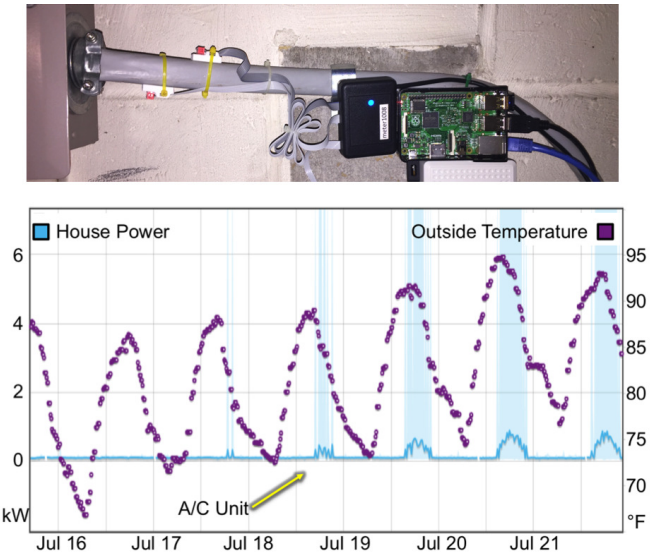

Fig. 11. (Top) A NILM installed on a split phase residential power line (Bottom) During a family vacation, power consumption is close to zero until the outdoor temperature increases beyond the thermostat set point of 80 degrees

.

The vessel goes underway at 07:45 and returns at 15:30. This is indicated by the temporary loss of power that occurs when the ship transitions between shore power and onboard generators. At large timescales the power data appears noisy but as indicated by the inset, the waveform is actually a series of distinct transients caused by loads switching on and off. Of particular note in this data set is the higher baseline power level after the ship returns to port. This indicates the crew did not properly turn off equipment before leaving the ship for the day.

Joule NILM's also make it possible to monitor power usage in low-margin environments where traditional current and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815432, IEEE Internet of Things Journal

voltage sensors are cost prohibitive. In particular these NILM's are an excellent tool for monitoring residential energy usage. Figure 11 shows results from a NILM installed on a single family home. The plot shows the energy usage during a week where the home was unoccupied.

The rise in energy usage from July 19-22 is caused by the AC unit. This could indicate a malfunctioning appliance, a broken window, or perhaps an unauthorized occupant. Adding a weather module to Joule explains the anomaly. The right axis is a temperature stream produced by a Reader Module that retrieves hourly weather data from a web API. The combination of temperature and power streams show that the AC unit was indeed operating correctly. The thermostat was set to $80°$F with the intention that it would not run while the house was unoccupied. However, the weather was warmer than expected which triggered the AC during peak outdoor temperatures. The NILM runs on a RaspberryPi 2 which operates at 35% load. The total bill of material for the installation is under $100 USD.

VI. Future Work

The network transport layer of Joule could be extended to support multipath routing, using, for example, the techniques presented in [20], [21]. This would improved the reliability of distributed pipelines in intermittent mobile networks. In addition to exploring new routing topologies we are also developing user interface (UI) design tools for visualizing the distributed datasets produced by Joule nodes.

VII. Conclusion

Joule is a modular framework for executing real time signal processing on low cost single board computers. Using Joule, high bandwidth sensor nodes can process data locally making them resilient to network outages and facilitating their integration with centralized IoT infrastructure. Using the client API, modules can be implemented in just a few lines of Python code. Modules are loosely coupled by data streams to form complex signal processing pipelines. Data streams use a common API regardless of their underlying transport mechanism. The flexibility provided by the Joule architecture introduces very little overhead, less than 4% for common workloads on the latest RaspberryPi SBC. Deployments on ships, homes, and labs running high bandwidth power monitors has proven Joule to be an efficient and reliable data processing framework.

References

[1] "Apache storm," http://storm.apache.org/releases/current/Rationale.html, accessed: 2017-11-30.

[2] D. Xiang, Y. Wu, P. Shang, J. Jiang, J. Wu, and K. Yu, "Rb-storm: Resource balance scheduling in apache storm," in *2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, July 2017, pp. 419–423.

[3] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, March 2015, pp. 154–161.

[4] "Apache kafka," https://kafka.apache.org/, accessed: 2017-11-30.

[5] M. Sewak and S. Singh, "Iot and distributed machine learning powered optimal state recommender solution," in *2016 International Conference on Internet of Things and Applications (IOTA)*, Jan 2016, pp. 101–106.

[6] L. Zhao, Z. Chuang, X. Ke-Fu, and C. Meng-Meng, "A computing model for real-time stream processing," in *2014 International Conference on Cloud Computing and Big Data*, Nov 2014, pp. 134–137.

[7] Y. Wu and K. L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 723–734.

[8] S. Kim, S. Kim, R. M. Kil, and H. Y. Youn, "Cgroup-aware load balancing in heterogeneous multi-processor scheduler," in *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct 2016, pp. 21–26.

[9] J. Paris, J. S. Donnal, and S. B. Leeb, "Nilmdb: The non-intrusive load monitor database," *IEEE Transactions on Smart Grid*, vol. 5, no. 5, pp. 2459–2467, Sept 2014.

[10] J. Donnal, J. Paris, and S. B. B. Leeb, "Energy applications for an energy box," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2016.

[11] "Numpy structured arrays," https://docs.scipy.org/doc/numpy-1.13.0/user/basics.rec.html, accessed: 2017-11-30.

[12] "Scipy: open-source software for mathematics, science, and engineering," https://docs.scipy.org/doc/scipy/reference/, accessed: 2017-11-30.

[13] I. Abubakar, S. N. Khalid, M. W. Mustafa, H. Shareef, and M. Mustapha, "An overview of non-intrusive load monitoring methodologies," in *2015 IEEE Conference on Energy Conversion (CENCON)*, Oct 2015, pp. 54–59.

[14] J. M. Alcal, J. Urea, . Hernndez, and D. Gualda, "Sustainable homecare monitoring system by sensing electricity data," *IEEE Sensors Journal*, vol. 17, no. 23, pp. 7741–7749, Dec 2017.

[15] J. C. Nation, A. Aboulian, D. Green, P. Lindahl, J. Donnal, S. B. Leeb, G. Bredariol, and K. Stevens, "Nonintrusive monitoring for shipboard fault detection," in *2017 IEEE Sensors Applications Symposium (SAS)*, March 2017, pp. 1–5.

[16] D. Egarter, V. P. Bhuvana, and W. Elmenreich, "Paldi: Online load disaggregation via particle filtering," *IEEE Transactions on Instrumentation and Measurement*, vol. 64, no. 2, pp. 467–477, Feb 2015.

[17] H. H. Chang, M. C. Lee, W. J. Lee, C. L. Chien, and N. Chen, "Feature extraction-based hellinger distance algorithm for nonintrusive aging load identification in residential buildings," *IEEE Transactions on Industry Applications*, vol. 52, no. 3, pp. 2031–2039, May 2016.

[18] J. M. Gillis, S. M. Alshareef, and W. G. Morsi, "Nonintrusive load monitoring using wavelet design and machine learning," *IEEE Transactions on Smart Grid*, vol. 7, no. 1, pp. 320–328, Jan 2016.

[19] J. Paris, J. S. Donnal, Z. Remscrim, S. B. Leeb, and S. R. Shaw, "The sinefit spectral envelope preprocessor," *IEEE Sensors Journal*, vol. 14, no. 12, pp. 4385–4394, Dec 2014.

[20] M. Z. Hasan, H. Al-Rizzo, and F. Al-Turjman, "A survey on multipath routing protocols for qos assurances in real-time wireless multimedia sensor networks," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1424–1456, thirdquarter 2017.

[21] M. Z. Hasan and F. Al-Turjman, "Optimizing multipath routing with guaranteed fault tolerance in internet of things," *IEEE Sensors Journal*, vol. 17, no. 19, pp. 6463–6473, Oct 2017.