Abstracting Distributed, Time-Sensitive Applications

Kyle Liang

CMU-S3D-25-120

September 2025

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
Carlee Joe-Wong
Joshua Sunshine
Aviral Shrivastava (Arizona State University)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

This material is based upon work supported in part by the National Science Foundation (awards 1646235 and 1901033), the Department of Defense (awards H9823018D0008 and H9823023C0274), the Air Force Research Lab with the Defense Advanced Research Projects Agency (BRASS award FA87501620042), and the U.S. Geological Survey (award 170655-23048/G24AC00117-00).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: programming languages, dataflow graphs, macroprogramming, case study, user study

Abstract

Distributed and Time-sensitive (DT) cyber-physical systems are challenging to design and develop. When writing systems in conventional languages, programmers struggle to write programs at scale due to the complexity of intrusive, cross-cutting concerns in a heterogeneous, distributed system with sensing and actuation timing requirements.

This thesis proposes that a system using a macroprogramming language with dataflow graph semantics can reduce barriers to entry for programmers and domain experts unfamiliar with designing distributed, time-sensitive applications. We present TTPython: a domain-specific language and runtime system designed to write shorter and cleaner code for challenging cyber-physical applications. Its novel, timed, tagged-token dataflow graph execution model allows programmers to develop applications at a macroprogramming scale while supporting timing specifications such as periodicity and soft deadlines. The programmer specifies timing requirements and uses decorators and system-provided function calls to guide TTPython in the placement of code while TTPython handles distribution, communication, and coordination between devices. We have evaluated TTPython by comparing it to a best-practice implementation of a 1/10th-scale connected autonomous vehicle application. An in-progress case study on an intercity flooding application examines how TTPython affects design and system decisions during development. These two case studies serve as the foundation of a user study in which we ask users to write these DT applications using either TTPython or vanilla Python with a message-broker system. We find that programmers struggle when writing infrastructure code that TTPython abstracts, and find TTPython straightforward and helpful.

Dataflow graphs have been applied in various contexts in parallelism and distributed computing, but little work has focused on token-tagged dataflow graphs. This thesis will show how incorporating time in a token-tagged dataflow graph makes it an effective tool for handling distributed, time-sensitive applications. We contribute key abstractions that make programming DT applications at scale approachable. The modified dataflow graph serves as the initial framework towards creating a standard digital distribution service for developing and deploying DT applications across shared devices, such as in a smart city.

Acknowledgments

First and foremost, I would like to thank my advisor, Jonathan Aldrich. He has always supported me through my tumultuous journey of the Ph.D. and guided me through difficult times. He had big shoes to fill when advising fell through, and I thank him for his strength and encouragement. Jonathan has helped me grow as an engineer, a writer, and a researcher. Although this project has quickly grown unmanageable, Jonathan helped me carve a piece out for me to feel proud of the work.

The members of my thesis committee have been instrumental in shaping my research. Aviral Shrivastava and Carlee Joe-Wong have been great mentors since the inception of my Ph.D. journey. Joshua Sunshine has been very helpful with my concerns on mentoring relationships and his insights on human studies with relation to PL and systems.

My prior research advisors and mentors have been a key influence for me to pursue a Ph.D. I want to thank Aydogan Ozcan, Steve Feng, Hatice Koydemir, Jens Palsberg, Christian Kalhauge, and Jason Teoh for the great experiences I've had during my undergrad.

Out of the many friends prior to the Ph.D., I'd like to give thanks to John Luu and Gauri Ratra. John has been with me through thick and thin, and I admire his tenacious outlook on life. He is like a brother to me; I somehow manage to collect all of his hand-me-downs. Gauri has been an absolute joy in my life. She has helped retain my sanity throughout my undergrad and during my Ph.D. as a source of fun and misery. Gauri perfectly embodies the phrase, "The grass is always browner on the other side."

Maybe the real Ph.D. was the friends I've made along the way. I can't imagine having better officemates than Eli Claggett and Peter Carragher. I've had great fun at board games with Sam Estep, Luke Dramko, Chenyang Yang, and Soham Pardeshi. The gym experiences were unparalleled with Jenny Liang, Kush Jain, Manisha Mukherjee, and Nadia Nahar. I wouldn't have picked up ping pong/pickleball without the help of Vasu Vikram, Parv Kapoor, Simon Chu, Will Epperson, Aidan Yang, and Alan Shen. Lunch and Learn would be much more difficult without Maria Casimiro, Long Nguyen, and Jenny Tang helping out. I can't forget my senior peers, including Jenna Wise, Wode Ni, Zeeshan Lakhani, and David Widder. And last but not least, Jane Hsieh and Leo Chen have always been there for me. I've truly loved the trips and experiences we've had, from international travel to movie nights and questionably named ramen.

Although undocumented in this thesis, it would be remiss not to thank the teaching faculty who have inspired and guided me. Thank you, Sol Boucher, for being an excellent mentor and a co-instructor for 15-122. Thank you, Iliano Cervesato, for guiding us both in that class. Thank you, Zack Weinberg, for your logistical mastery and for being a co-instructor for 15-213. And finally, thank you, Michael Hilton, for inspiring me to pursue education in my future endeavors.

Last but not least, my family has always been part of my core supporters. I can't thank my dad, Ching-Ming Liang, enough for being there when I needed him. My mother, Hwei-Mei Liang, has always given great advice (sometimes unsolicited), which I always appreciate. And to my sister, Justine Liang, who will always be the fakest (real) doctor to me.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

1.1 Introduction

Distributed, time-sensitive (DT) applications are ubiquitous in cyber-physical systems (CPS). Improvements in wireless, embedded, cloud, and networking technologies enable larger, more interconnected applications that measure and control the physical world. For example, large drone swarms determine their location and plan their path at millisecond or microsecond precision to form spectacular displays like Intel's swarm at the 2018 Winter Olympics in South Korea. Recent research is working towards autonomous vehicular networks with signal-free intersections [32] in which vehicles coordinate their trajectories through the intersection to use the space more efficiently and increase throughput. The vehicles must plan these trajectories with millisecond precision to avoid colliding with each other.

When writing DT applications in conventional languages, programmers struggle to manage the complexity of handling time as a control flow concept. Time-based control flow is difficult to implement. Three common examples are periodicity, watchdog timers, and data synchronization. Programmers encode periodic tasks with infinite for-loops that poll the system clock to wait until the correct time. Watchdog timers depend on concurrent execution to interrupt the main process if the time allotted has passed. Data synchronization means ensuring that computation occurs with data aligned in time. For example, a vehicle in a smart intersection uses asynchronous processes to control the LIDAR and camera and needs to ensure that incoming data is synchronized for execution.

We present a sample best-practices application written in Python for the smart intersection in Figure 1.1 to illustrate these control-flow concepts. Lines 13 and 14 depict periodicity, while Lines 19 and 24 show watchdog timers. Line 20 represents data synchronization, where the processes will periodically generate data and remove stale data. The camera process is shown from Lines 3 to 5. In general, these control schemes mimic timed execution with unintuitive or asynchronous control mechanisms because current control schemes execute without respect to time.

Time-based control flow is also difficult to specify. For example, a car in a smart intersection depends on the agreement of object locations in its vicinity with nearby vehicles and infrastructure to increase traffic throughput. A complete description of its timing specification involves

the car listening to all nearby devices before making a decision. This involves sending its locale information over the network to other constituent cars and infrastructure devices in the area. A car should use a watchdog timer to gracefully degrade (*i.e.* slow down) if it does not hear back. However, this specification is split by implementation when programmers need to specify global timing requirements across files and separate applications, which increases code complexity and can cause bugs. For example, split implementations led to a bug where the `checkin` function on Line 24 uses a GET HTTP request with a timeout far longer than intended (1 second vs 100 milliseconds), which was fixed in our system. Furthermore, the `checkin` function's receiving device for the implicitly assumed that the last GET HTTP request it received from each vehicle represented its most up-to-date position. GET requests are not guaranteed to arrive in order, which can lead to a synchronization issue. Overall, the separation between time and data in languages designed without DT systems in mind leads to verbose and convoluted code.

```
1  def sourceImagesProcess(...):
2      # get cam_data
3      while not out_queue.empty(): # clear old data
4          out_queue.get()
5      out_queue.put(cam_data) # in-order send
6
7  # Start after 10 seconds
8  start_time = time.time() + 10_000_000
9  interval = 125_000 # Interval is 125ms
10
11 # Spawn lidar and camera processes
12 target = start_time
13 while 1: # infinite loop
14   if (target <= time.time()): # polls rapidly
15     now = time.time()
16     lidar_received, camera_received = False, False
17
18     fallthrough = time.time() + fallthrough_delay
19     while (time.time() < fallthrough and not # polls
20         (lidar_received and camera_received)): # sync
21         # Get lidar_received and camera_received
22     if lidar_received and camera_received:
23         # send coordinates to global model
24         response_message = rsu.checkin(...) # blocking
25         if response_message == None:
26             coast()
27         else:
28             # normal execution
29     else:
30         emergencyStop()
31
32     target = target + interval
33   time.sleep(.001)
```

Figure 1.1: *A best-practices autonomous vehicle implementation of the smart intersection.*

We define **time-sensitivity** in applications for cyber-physical systems as having the following features: periodically generating data asynchronously, synchronizing data, having timing requirements across devices, and responding in the event of failure through time-triggered exception code.

We present TTPython, a deeply embedded domain-specific language ([55]) and runtime embedded in Python. Its design goals are to help programmers write shorter and cleaner DT applications. The programmer writes their application in a single file while adding decorators and system calls to specify distribution and timing requirements. TTPython then handles the distribution, communication, and coordination between devices to realize this. TTPython's abstractions for distribution are inspired by macroprogramming frameworks [6, 7] for selecting sets of devices, efficient in-network aggregation, and interfaces between heterogeneous devices in the system. Programmers can then specify time-triggered exception handling (Plan B) [33] across devices.

Although we did not develop the first iteration of TTPython, we have expanded on the scope of timing specifications in TTPython to incorporate time-triggered exception handling. We find that time-triggered exception handling is a minimum requirement for many DT applications. This thesis also provides and implements the semantic execution model of TTPython. To inte-

grate macroprogramming into a conventional host language, we compile code to a dataflow graph (DFG) architecture, which is composed of nodes of computation and edges for data communication. The DFG intermediate representation adapts the MIT Tagged-Token Dataflow Architecture (TTDA) [51] to the DT setting. The DFG provides abstractions for concurrency and communication, and these abstractions offer maximum flexibility to partition the program across devices to maximize efficiency or take advantage of hardware resources. TTPython distinguishes itself from prior literature by introducing time in the dataflow model. Data is encapsulated in tokens with a tag to indicate the liveness of the value. The tag includes a time interval to indicate which iteration of periodic execution the data belongs to. An extra control plane added to the TTDA allows us to encode periodic and deadline behaviors at the graph level; thus, the programmer can specify timing at the global system view. To our understanding, we are the first to explore a time-based dataflow architecture.

1.2 Thesis Statement

The use of a macroprogramming language with timed, tagged-token dataflow graph semantics enables users to write shorter and more robust networking, timing, and concurrent code in distributed, time-sensitive applications.

To evaluate this, we examine TTPython through two case studies and a user study.

1.3 Evaluation

We apply TTPython on two distributed, time-sensitive applications: a 1/10th-scale smart intersection with connected autonomous vehicles (CAV) and an urban flooding sensor network. These applications vary in their time and distribution requirements. These case studies capture different insights into TTPython. The smart intersection gives us comparable code artifacts by comparing a DT application written in both TTPython and vanilla Python. We compare challenging timing and distribution code that TTPython abstracts, present the amount of code reduced by these abstractions, and run both applications to identify bugs and differences in the implementations. The urban flooding network provides an additional data point of TTPython's flexibility in describing DT applications. Its use cases show how TTPython primitives can describe complex timing requirements. These two case studies shed light on how TTPython affects design and development and the challenges found in using TTPython.

These case studies were used as inspiration for task settings for a qualitative user study on TTPython. Its purpose is to highlight the difficulties programmers face when using conventional frameworks to write DT applications. Participants wrote these applications with both TTPython and vanilla Python to identify the advantages gained with the abstractions TTPython offers. Each application is broken down into five tasks for users to complete within an hour and a half. Participants were able to run and test their code to check if their implementation matched the required specifications. We collected a mix of quantitative and qualitative data: time spent programming, bugs introduced, self-reported usability and mental load, and transcripts on perceived difficulty. We found that participants struggle with recreating TTPython's timing and distribution abstrac-

tions in vanilla Python. Participants reported that it was easier to manage isolated, cross-cutting concerns; however, they also reported difficulty when composing distribution and timing requirements in TTPython. These reports indicate that TTPython's success is linked with how well the application's system architecture is understood and specified.

1.4 Contributions

This thesis contributes the following work:

- Creating syntax for the time-triggered exception handling code (watchdog timer) and its corresponding semantics in TTPython.
- Providing compilation and operational semantic rules for the timed, tagged-token dataflow graph.
- Analyzing case study code and design patterns.
- Creating and running a qualitative user study on TTPython.

1.5 Outline

We first provide a high-level description of the TTPython system along with its syntax, compilation, and dataflow graph semantics in Chapter 2. Section 3.1 describes a case-study in which we evaluate the experience of designing a non-trivial DT application of a 1/10th-scale intersection with connected autonomous vehicles (CAV). The Urban Flooding application is covered in Section 3.2 and describes common design patterns observed when practitioners use TTPython. The work culminates with a comparative user study in Chapter 4 to observe how TTPython abstractions benefit programmers compared with other classical solutions. Chapter 5 describes more in-depth description of research that inspired TTPython.

Chapter 2

TTPython

2.1 Problem and Approach

We identified four issues to define time-sensitive applications for cyber-physical systems. These applications periodically generate data asynchronously, synchronize data, have timing requirements across devices, and respond to failures through time-triggered exception handling. To address these issues, we created TTPython, a domain-specific language embedded in Python. TTPython integrates physical time with data. The insight is that data needs time in DT applications. Data has a notion of "liveness" in that two values are valid to use together if they are "close" enough in time. Consider an example of taking an average of temperature samples within a building. Generally, the temperature of a building does not drastically change within a five-minute interval. Temperature samples that differ within five minutes could be used together to calculate an average ambient temperature. However, day and nighttime temperatures can vary widely. Averaging temperatures taken hours apart might not be meaningful. The environment's characteristics define these time-liveness intervals. This forms the idea of data synchronization. Data that is generated around the same time can be used together. Data synchronization appears in TTPython as an execution rule for our dataflow graph.

TTPython's time model uses wall-clock time. When generating data, each device will take its local Unix time to generate timestamps. Time in TTPython appears as data (such as when specifying a deadline) or as a context associated with data (its "liveness"). This context is defined as a time interval composed of two Unix timestamps. These time intervals allow us to define what "close" enough is and mitigate inaccuracies caused by jitter and desync.

We first see how TTPython abstracts asynchronous data generation and data synchronization through function decorators (`@STREAMify` and `@SQify`) by walking through a "Hello-World" style application. We then add multiple devices to create timing requirements across devices and add deadlines so they can respond in the event of failure. These are handled in TTPython by system-provided context managers and functions (`TTConstraint` and `TTFinishByOtherwise`. The `TTConstraint` enables TTPython macroprogramming design and allows global timing specifications to be expressed without needing to break them down as individual timers per device.

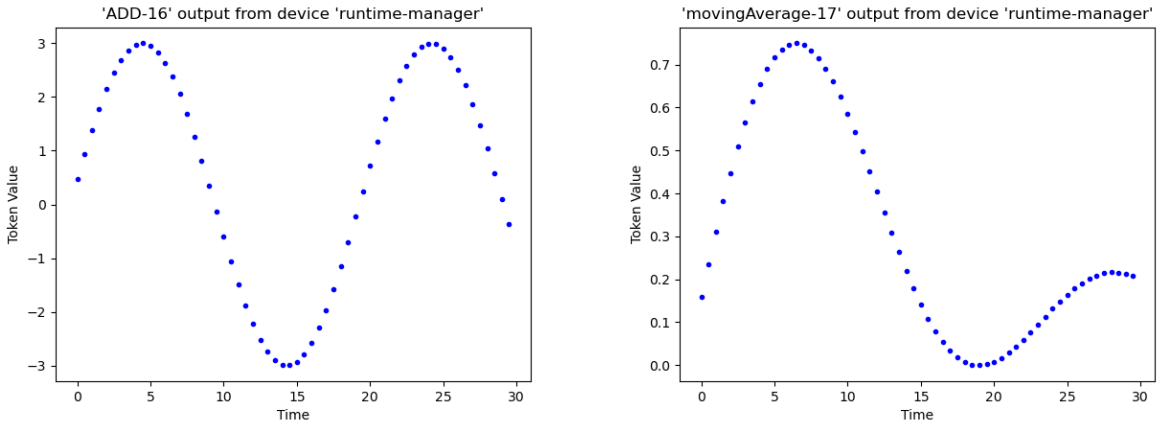
Although TTPython shares syntax with Python, its execution differs based on the level of

abstraction the programmer writes in. The `@GRAPHify` decorator designates a main function in which its body is converted to a dataflow graph. However, non-main function bodies execute with ordinary Python semantics familiar to programmers. TTPython wraps these functions as separate nodes in the dataflow graph and provides infrastructure code, which includes the necessary data synchronization and networking between the functions in the main body. To clarify when TTPython runs infrastructure code and when user code executes, we will first introduce the compilation of TTPython syntax to the dataflow graph. This also shows how the dataflow graph incorporates user-specified timing specifications, such as periodicity. We then describe its operational semantics, describing how a node's execution rules enforce data synchronization or run time-triggered exception handling code.

2.2 TTPython System Overview

2.2.1 Example TTPython Program

We introduce the language with a sample sensor fusion application adding two simulated periodic sine wave data sources as seen in Listing 1. For 30 seconds, the application adds data from the sine waves (Line 44) and keeps track of the average of one of the sine waves for 30 seconds (outputs depicted in Figure 2.1).



(a) The sum of two sine waves.

(b) The moving average of one sine wave.

Figure 2.1: Output of the sample sensor fusion app in Listing 1.

To create a TTPython program, the programmer first specifies the main function with the function decorator `@GRAPHify`, as seen on Line 20. `@GRAPHify` takes the associated function and constructs a dataflow graph from its body. It requires that all expressions within the function must be basic arithmetic, boolean expressions, or TTPython decorated functions with `@SQify` or `@STREAMify`. The compiler automatically takes basic Python arithmetic and boolean operators and converts them to their SQ (a node in the graph) counterpart. We use the term SQ (scheduling quanta) to distinguish the timing characteristics it requires from its nodal counterpart in prior literature.

```python
@STREAMify
def sinusoid_sampler(A, f, phi):
    from math import sin, pi  # local import
    global sq_state
    if sq_state.get('count', None) is None:
        sq_state['count'] = 1
    sample = A * sin(sq_state['count'] * 2 * f / pi + phi)
    sq_state['count'] += 1
    return sample

@SQify # user defined SQ
def average(new_input):
    global sq_state           # persistent local state
    count = sq_state.get('count', 0)
    avg = (sq_state.get('avg', 0) * count + new_input) / (count + 1)
    sq_state['count'] = count + 1
    sq_state['avg'] = avg
    return avg

@GRAPHify # main program specifying SQ linking
def add_sine(trigger):
    A_1 = 1; f_1 = 0.25; phi_1 = 0
    A_2 = 2; f_2 = 0.10; phi_2 = 0

    with TTClock.root() as root_clock:
        start_time = READ_TTCLOCK(trigger, TTClock=root_clock)
        N = 30 # seconds
        stop_time = start_time + (1_000_000 * N)

        # create a sampling interval by setting the
        # start and stop tick for one of the args
        sampling_time = VALUES_TO_TTTIME(start_time, stop_time)
        A1_sample = COPY_TTTIME(A_1, sampling_time)
        A2_sample = COPY_TTTIME(A_2, sampling_time)
        sine_1 = sinusoid_sampler( # streamify call 1
                    A1_sample, f_1, phi_1,
                    TTClock=root_clock, TTPeriod=500_000,
                    TTPhase=0, TTDataIntervalWidth=100_000)
        sine_2 = sinusoid_sampler( # streamify call 2
                    A2_sample, f_2, phi_2,
                    TTClock=root_clock, TTPeriod=500_000,
                    TTPhase=0, TTDataIntervalWidth=100_000)

        output = sine_1 + sine_2    # sync data streams
        y = average(output)
        return output
```

Listing 1: A TTPython application adding two sine waves and tracking the average periodically every 0.5 seconds for 30 seconds.

The GRAPHified function `add_sine` is populated by SQs created with the function decorators `@SQify` and `@STREAMify`. `@SQify` takes the associated function and translates it into a node in the dataflow graph intermediate representation. Line 11 shows `@SQify` applied to the average function. It weights the existing average by the number of samples, adds the new sample value, and divides by the new number of samples. The average implementation requires tracking the number of samples taken between iterations. To account for this, TTPython reserves and overloads the global variable `sq_state` to store local state between multiple executions of a SQ. Note that this does not share semantics with Python's **`global`** keyword. Instead, the variable is only locally observable by the SQ, so accesses are to the local device's version of the SQ's state. TTPython does not support global memory space: the data passed between SQs are copied.

The `@SQify` and `@GRAPHify` constructs we have described so far are insufficient for describing computation called at fixed intervals of time (*i.e.* periodic computation). The function decorator `@STREAMify` creates a SQ that will repeatedly generate data once started. In short, the STREAMified function will periodically execute while all its inputs are "live." In the DFG, a SQ communicates data across edges through *tokens*. Tokens pair time intervals with data. A computation involving multiple data values can execute if these data share overlapping time intervals, indicating that they share a common temporal context. We modify a token's time interval with internal functions TTPython provides to modify token structure. `VALUES_TO_TTIME` sets the start and end timestamp of the output token's time interval to be the values from the two input tokens. `COPY_TTIME` creates a new token with the value of the first argument and the time interval of the second argument. On Line 27, by setting N=30, we specify that the stream is allowed to emit data for the specified time interval of 30 seconds. These modifications allow us to specify how long a STREAMified function will run. A STREAMified function will generate output tokens periodically until the current time no longer overlaps with the time intervals of its input tokens. Concretely, in Listing 1, we set `A1_sample` and `A2_sample` to have a time interval of `start_time` and `start_time + 30` seconds. The function will start running periodically at `start_time` and stop executing periodically when 30 seconds pass since its start.

The call to `sinusoid_sampler` function needs periodicity information via specially named keyword arguments `TTClock`, `TTPeriod`, `TTPhase`, and `TTDataIntervalWidth`. The `TTClock` specifies the degree of time synchronization across devices. It is a placeholder and relates to hardware clock primitives that this thesis will not discuss. The programmer can specify how often the SQ will trigger with `TTPeriod`. A `TTPhase=0` means that the `sinusoid_sampler` will trigger when an internal clock counter reads 0 modulo the period; for `TTPeriod=500000`, a `TTPhase=250000` would trigger 250ms after the counter wraps around and every 500ms successively after (as a 500ms period was specified). When running periodic computation, the SQ also needs to define the time interval for its generated tokens. When the runtime executes this SQ, the runtime will take the start and stop time for executing this SQ and take the average. The new interval is this timestamp average, plus-minus the `TTDataIntervalWidth` divided by 2. The width of the resulting interval is thus `TTDataIntervalWidth`. TTPython reserves keyword arguments that start with `TT` as information used by the runtime and are not accessible by the function body during execution.

```
1 @SQify
2 def deadline_check():
3     return None
4
5 @GRAPHify
6 def add_sine(trigger):
7     ...
8     with TTClock.root() as root_clock:
9         ...
10        deadline_time = READ_TTCLOCK(sine_1,    # deadline time
11                                     TTClock=root_clock) + 50_000
12        ...
13        with TTConstraint(name="dev"): # assign to device 'dev'
14            safe_add = sine_1 + sine_2
15
16        safe_add = TTFinishByOtherwise(output,
17                                       TTTimeDeadline=deadline_time,
18                                       TTPlanB=deadline_check(),
19                                       TTWillContinue=True)
20        # if deadline fails, it produces a separate
21        # value (similar to ternary operation:
22        # a = y if clock.now < t else deadline_check())
23        average(safe_add)      # downstream SQ
```

Listing 2: An updated example of the deadline construct with location specification with adding two sine waves.

2.2.2 Time and Location Constraints

TTPython also offers two more constructs at the `@GRAPHify` level to assist programmers in dealing with space and time constraints. We modify our sinusoid program in Figure 2 to include a deadline constraint on Line 23 for adding the two sine wave values and specifying the location of the addition SQ to a particular device on Line 13.

Specifying Location

Before graph execution, TTPython needs to decide how to map each SQ in the dataflow graph to its host device. We offer a simple constraint syntax with the keyword `TTConstraint` where the programmer can specify requirements on groups of SQs. These requirements can range from required hardware for an SQ (*e.g.*, a function that takes an image requires a camera on the device) or software (*e.g.*, a specific library for Fast Fourier Transforms). In this program, the programmer has specified that the SQ within the **`with`** block on Line 13 will be assigned to the device named "dev". If multiple devices fulfill a requirement, an arbitrary device is selected. Future work will focus on optimizing this selection with respect to timing and other user-specified parameters.

Deadline Constructs

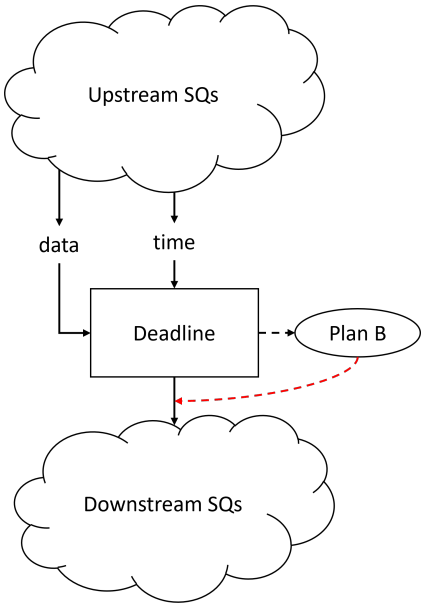


Figure 2.2: Deadline DFG Compilation

Timing requirements in TTPython have two components: the time when the computation should finish and the exceptional time handling code to run if the execution doesn't finish on time. In the updated example in Figure 2, `deadline_time` is generated and sent downstream by each iteration of `sinusoid_sampler`. The TTPython SQ `TTFinishByOtherwise` construct ensures that data has been generated on time by triggering a backup procedure (*Plan B* [33]) if the data is not received by the specified deadline, which is `deadline_check` in our example. If Plan B is run, the programmer can specify different behaviors regarding downstream execution. These behaviors are similar to those associated with raising an exception. The timestamp generation comes from `READ_TTCLOCK` on Line 10 and offsetting it with an addition. `READ_TTCLOCK` accepts two parameters: a trigger value that upon token arrival will read the clock and a clock from which to take the timestamp. TTPython uses a clock to allow programmers to specify the level of synchronization necessary between time-sensitive actions. The root clock, a proxy for Universal Coordinated Time, is sufficient for the examples in this thesis.

The SQ `TTFinishByOtherwise` supports two recovery behaviors: replacing a "late" value with a default value provided by Plan B or running a separate backup routine without executing the rest of the data dependencies of that data variable. For example, a backup routine on an autonomous vehicle that instructs the car to apply the brakes may not want downstream SQs to execute after applying the brakes, as they could command the car to continue moving forward. `TTWillContinue` accepts a boolean and indicates whether the programmer intends to execute the downstream SQs from the data value given if Plan B is executed. If `TTWillContinue` is **`True`**, then the default value passed on will be the value returned by the function call specified with `TTPlanB`. Downstream SQ execution will continue with this default value. This pass-through procedure is visually represented in Fig 2.2 by the red dotted line. Otherwise, if `TTWillContinue` is **`False`**, the compiler will not include the red dotted line. The boolean value **`True`** for `TTWillContinue` in Figure 2 indicates that the graph will always execute the `average` SQ on line 23 regardless if Plan B occurred. The graph will use the value **`None`** from the output of `deadline_check` as the value for `safe_add` if `sine_1` does not arrive by `deadline_time`.

2.2.3 Composing Time Constructs

Having data synchronization (i.e. all tokens have overlapping time intervals) as a primitive allows TTPython to describe interesting timing and control schemes. For example, consider the following code in Listing 3.

The function `lepton_planb` on Line 2 is used to reset a Thermal Imaging Camera FLIR

```
1 @SQify
2 def lepton_planb(): # resets the
3     ...
4     # set low for 5 seconds for pin 31
5     # to trigger reset on breakout board
6     GPIO.output(pin, GPIO.LOW)
7     sleep(5.0)
8     # reset to default state
9     ...
10    return {}
11
12 @GRAPHify
13 def main(trigger):
14    ...
15    with TTClock.root() as root_clock:
16        ...
17        lr = lepton_record(sample_window,
18                           TTClock=root_clock,
19                           TTPeriod=3_000_000, # periodic of 3 secs
20                           TTPhase=0,
21                           TTDataIntervalWidth=250_000)
22        lepton = TTFinishByOtherwise(lr,
23                          TTTimeDeadline=timestamp + timeout_val,
24                          TTPlanB=TTSingleRunTimeout(  # prevents livelock
25                              lepton_planb(), TTTimeout=10_000_000),
26                          TTWillContinue=False)
```

Listing 3: This example composes the deadline and data synchronization constructs by only allowing one Plan B (`lepton_planb`) to run at a time.

Lepton 3.5. As a cost-effective thermal camera, the Lepton sometimes becomes unresponsive. An effective measure to recover from this hardware fault is to turn it off and on [34]. The procedure to reset the Lepton 3.5 requires about five seconds time to cycle its power. However, the Lepton is being called periodically every three seconds with `TTPeriod=3_000_000` on Line 19. Without the `TTSingleRunTimeout` call enclosing the Plan B expression, the graph will have livelock where calls to Plan B will be repeatedly called without chance to recover. For example, when the Lepton first becomes unresponsive, Plan B will be called. As the Plan B node takes five seconds to complete, it will finish before the next periodic iteration starts. The next periodic iteration then attempts to wait for Lepton, which fails to respond because it is in the process of restarting. This then creates another call to Plan B, which then spawns a new execution attempting to power cycle the thermal camera. This leaves the graph in an unrecoverable state of trying to reset the camera with Plan B but not letting enough time elapse for it to run to completion.

To address livelock issues, TTPython provides the `TTSingleRunTimeout` function call. This construct takes an expression as input and a keyword argument `TTTimeout`. It prevents the expression from being evaluated more than once and waits for `TTTimeout` microseconds before allowing the expression to be evaluated again. The construct introduces a dedicated token used

to synchronize the expression it encapsulates. This token acts similarly as a mutual exclusion for the expression, as the SQ that uses this token needs to synchronize with this token and ensures that this token is regenerated only generated once the expression completes. Its time interval is initially set from [-∞, ∞] to overlap with any interval. When `lepton_planb` is first called, the mutex token is consumed with the trigger to the `lepton_planb` SQ. This data synchronization step prevents other Plan B executions from occurring as other call attempts would lack this mutex token. After `lepton_planb` produces a value, `TTSingleRunTimeout` needs to regenerate the mutex token. It takes the time it finished execution as `curr_t`, the value in `TTTimeout` as n, and sets the mutex token with time interval as [`curr_t` + n, ∞]. Because the mutex can only match with tokens that are generated after the Plan B execution with a delay specified by `TTTimeout`, this prevents livelock by ignoring prior requests to reset the Lepton while it is already in the process of resetting.

2.3 Timed, Tagged-Token Dataflow Compilation

To gain an intuition on how tokens travel through the graph, we first look at how the graph is constructed. This gives insight on how function decorators and compound instructions (such as `TTFinishByOtherwise` and `TTSingleRunTimeout`) are represented in the graph.

We first briefly introduce the original MIT TTDA architecture. The TTDA is a graph composed of nodes of computation and edges as communication links between nodes. A sample dataflow graph node is shown in Figure 2.3. Data travels through the edges as a token. The MIT TTDA supports parallel architectures and can execute functions concurrently. It can do so by tagging tokens with a context, which is used to identify which data belongs to the same execution. In TTPython, this context is a time interval.

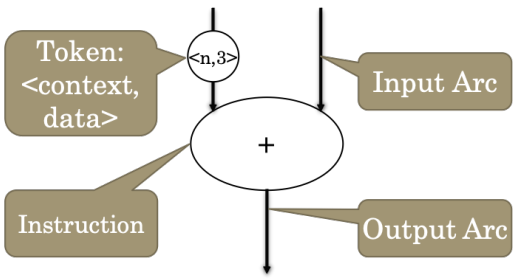


Figure 2.3: A dataflow graph node containing a + operation.

A SQ *fires* (executes computation) when certain conditions are satisfied, known as its **firing rule**. TTPython has three firing rules: the Data-Validity (DV), the Time-Based Trigger (TBT), and Immediate. The DV is the default firing rule for most SQs as it synchronizes data. The TBT enables the DFG to encode deadlines. The Immediate rule allows merging outputs from conditional branch computation. If the firing rule is successful, the SQ then applies its functional computation to the data and updates the token's context for downstream SQ forwarding. Tokens are stored in a *waiting-matching section*, where tokens wait until the SQ accumulates enough tokens with the same context to fire. We implement this as a set that accumulates tokens for each input port.

2.3.1 Definitions

The dataflow graph $\mathbb{G}$ is represented by a set of scheduling quanta V. Each SQ has a set of named input ports and a set of output ports. Edges are implicitly defined by the output ports, which are represented as a list of the names $\overline{\mathtt{x}}$ of the input ports they are connected to.

A SQ ($v \in \mathsf{V}$) is defined as $\langle \overline{o} = f(\bar{i}), r, s \rangle$ where f is the function v is encapsulating, $\bar{i}$ is a list of input ports, $\overline{o}$ is a list of output ports, r describes the firing rule for v, and s is the internal state of the SQ. Input port names do not have to be unique. In our implementation, we derive these names from source variables that may be used multiple times. Therefore, all input ports that share a name will be delivered the same token.

f is of type $\mathsf{List}\langle\mathsf{Token}\rangle \rightarrow \mathsf{Set}\langle\mathsf{Pair}\langle\mathsf{Token}, \mathsf{Name}\rangle\rangle$, where the name is the name of input port(s) we are sending the corresponding token to. We assume the implementation of the function does this by looking up what the SQ's output ports are connected to. This abstraction gives a function the option of choosing to generate an output token for each of its output ports. This flexibility enables conditional branching.

Each input port has a set $\mathtt{W}$, which holds tokens received by the port until the SQ fires. The structure of an input port i is composed of $\langle \mathtt{x}, \mathtt{W} \rangle$. s is the internal state of the SQ for firing rule purposes and differs from the functional `sq_state` as described in Section 2.2.1. This is used to keep track of particular tokens for the Time-Based Trigger firing rule, which we will discuss later in Section 2.4.3.

Tokens

An execution of the graph involves tokens propagating through $\mathbb{G}$. A token k is defined as follows:

$$k = \langle d, t \rangle$$

where d is data and t is a time interval $[t_s, t_e]$ where $t_s \leq t_e$. This timestamp represents the context of the data. This data was generated at $(t_s + t_e)/2$ and can be used in computation with other data that shares the same context (*i.e.* timestamps overlap). TTPython's implementation uses physical (Unix) time for each timestamp in the time interval. We also use time during execution to check if a deadline has passed. This requires keeping track of what is current time and going forward in time. To simplify our semantic model with these operations, we represent t as elements of $\mathbb{N}$.

By default, an output token's time interval is the intersection of all the input tokens' time intervals. If the programmer needs to override this default, such as extending the lifetime of a value beyond that of the data it was computed from, we provide APIs for this purpose. These APIs are also used to set the time interval for tokens that specify the length of time to generate a periodic stream, as was shown in Figure 1.

2.3.2 Compilation Structure

We now describe the translation of TTPython source code to a dataflow graph. The translation rules we use are described below:

$$e \mapsto \mathsf{V}; x$$

e is an expression that compiles to a set of SQs V. x is the name of the output port that a future input port can be renamed to to receive tokens from the expression e.

Every dataflow graph must have at least one input. We reserve the name $\mathbf{r}$ as one of the required inputs to the graph and use it as the initial trigger for the graph.

We assume that source variables have been defined before use. If so, the output port providing the variable has been created either as an initial input to the graph or by another SQ. Referencing a variable returns the name of the connector providing that value.

Compiling Expressions

Accessing a variable means accessing the named connector providing the data name in the graph. The compilation returns the variable name.

$$\frac{}{y \mapsto \{\}; y}\ \textit{var} \tag{2.1}$$

Getting a numeric or boolean value creates a SQ emitting that value. As it has no data dependencies, the initial trigger $\mathbf{r}$ signals the start of its execution. The expression is first assigned a fresh variable to describe the local value it produces, which may get renamed during variable assignment. The `CONST` function takes a trigger token as input and creates a token with n as its data field and $[-\infty, \infty]$ as its time interval.

$$\frac{n : \mathsf{Val} \quad x \text{ fresh}}{n \mapsto \{\langle [x] = \texttt{CONST}(\mathbf{r}, \texttt{const} = n), \texttt{DV}, \cdot \rangle\}; x}\ \mathsf{Val}^1 \tag{2.2}$$

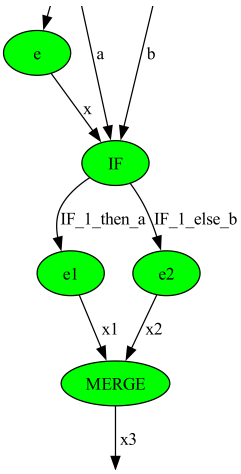
Arithmetic and boolean operations take two inputs, the left-hand side and right-hand side of the operator. Their outputs are then supplied to the input of the operator SQ.

$$\frac{e_1 \mapsto \mathsf{V}_1; x_1 \quad e_2 \mapsto \mathsf{V}_2; x_2 \quad x_3 \text{ fresh}}{e_1 \bigoplus e_2 \mapsto \mathsf{V}_1 \cup \mathsf{V}_2 \cup \{\langle [x_3] = \bigoplus(x_1, x_2) \rangle, \texttt{DV}, \cdot\}; x_3}\ \textit{Arithmetic} \tag{2.3}$$

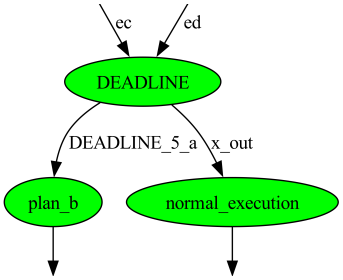
An If-Then-Else expression in Python evaluates e and executes e_1 if e's value is `True` or e_2 otherwise. We do not want to run both e_1 and e_2 without first checking the value from e because e_1 and e_2 might have side effects. Therefore, we create v_{then} and v_{else} that only allow their respective subgraphs to execute depending on the test expression's value. v_{then}'s function (`branch`) takes as input the test expression and the variables used in the then branch (*i.e.* free in) and forwards them to the subgraph only if the first token's data value is true. v_{else} is handled symmetrically with the `not_branch` function. This ensures that only one of the branches will execute.

After creating v_{then}, the connectors that the variables in the then branch reference should link with v_{then}'s output. Thus, v_{then} shadows the then branch's free variables' prior connectors. The variable substitution operation θ implements this shadowing.[2] We also shadow input triggers for SQs generated with Rule Val. The merge SQ takes any of its inputs and emits it as its output.

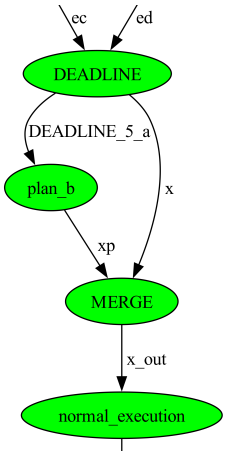
[2]Substitution: $e \circ \theta$, where $\forall(x \to x') \in \theta.\ e[x \to x']$



(a) Graph compilation of Rule If-Then-Else where the if branch and else branch use the free variables a and b respectively. v_{then} and v_{else} are represented by the **IF** node.



(b) Graph compilation of Rule Deadline Raise. $v_{deadline}$ and v_{planb} are represented by the **DEADLINE** node.



(c) Graph compilation of Rule Deadline Catch. $v_{deadline}$ and v_{planb} are represented by the **DEADLINE** node.

Figure 2.4: Compilation graphs for *If-Then-Else* and *Deadline*.

$$\frac{\begin{array}{c} e \mapsto \mathsf{V}; x \\ l_1 = \text{free}(e_1) \\ l_2 = \text{free}(e_2) \\ \theta = \{y \to y' \mid y \in l_1 \cup l_2\} \cup \{\mathbf{r} \to \mathbf{r}'\} \quad y' \text{ fresh} \quad \mathbf{r}' \text{ fresh} \\ v_{then} = \{(\text{list}(l_1) + [\mathbf{r}]) \circ \theta = \texttt{branch}(x, \text{list}(l_1)), \texttt{DV}, \cdot\} \\ v_{else} = \{(\text{list}(l_2) + [\mathbf{r}]) \circ \theta = \texttt{not\_branch}(x, \text{list}(l_2)), \texttt{DV}, \cdot\} \\ e_1 \mapsto \mathsf{V}_1; x_1 \\ e_2 \mapsto \mathsf{V}_2; x_2 \\ v_{out} = \{[x_3] = \texttt{merge}(x_1, x_2), \texttt{Immediate}, \cdot\} \quad x_3 \text{ fresh} \end{array}}{e_1 \texttt{ if } e \texttt{ else } e_2 \mapsto \{v_{then}, v_{else}, v_{out}\} \cup \mathsf{V} \cup (\mathsf{V}_1 \circ \theta) \cup (\mathsf{V}_2 \circ \theta); x_3} \textit{ If-Then-Else} \tag{2.4}$$

`TTFinishByOtherwise` follows a similar collecting and shadowing procedure for its Plan B subgraph with the v_{planb} node. The `test` function will send the data token as output on x or a token with value **`True`** on x_b depending on whether data comes in on time or Plan B needs to run. The SQ containing the `test` function uses the Time-Based Trigger firing rule and the SQs state, which will be discussed in below in Section 2.4. x corresponds to an on-time execution while x_b will start the Plan B subgraph. When `TTWillContinue` is **`False`**, TTPython will not substitute the value with the output from Plan B. Plan B will run as a side effect, and downstream execution will not happen as it will lack the data token necessary to execute.

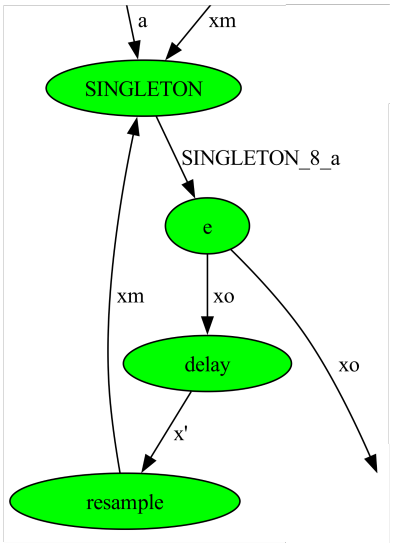
$$\frac{\begin{array}{c} e_d \mapsto \mathsf{V}_d; x_d \\ e_c \mapsto \mathsf{V}_c; x_c \\ l = \text{free}(e_p) \\ \theta = \{y \to y' \,|\, y \in l\} \cup \{\mathbf{r} \to \mathbf{r}'\} \quad y' \text{ fresh} \quad \mathbf{r}' \text{ fresh} \\ x \text{ fresh} \quad x_b \text{ fresh} \\ v_{deadline} = \langle [x, x_b] = \texttt{test}(x_d, x_c), \texttt{TBT}, \langle\langle -\infty, [-\infty, -\infty]\rangle\rangle\rangle \\ v_{planb} = \langle (\text{list}(l) + [\mathbf{r}]) \circ \theta = \texttt{branch}(x_b, \text{list}(l)), \texttt{DV}, \cdot\rangle \\ e_p \mapsto \mathsf{V}_p; x_p \end{array}}{\begin{array}{c} \texttt{TTFinishByOtherwise}(e_d, \texttt{TTTimeDeadline=}e_c, \\ \texttt{TTPlanB=}e_p, \texttt{TTWillContinue=False}) \mapsto \\ \mathsf{V}_d \cup \mathsf{V}_c \cup (\mathsf{V}_p \circ \theta) \cup \{v_{deadline}, v_{planb}\}; x \end{array}} \textit{ Deadline Raise} \tag{2.5}$$

When `TTWillContinue` is **`True`**, TTPython will substitute the value with the output from Plan B. A merge SQ allows Plan B's output to replace the data token for downstream execution.

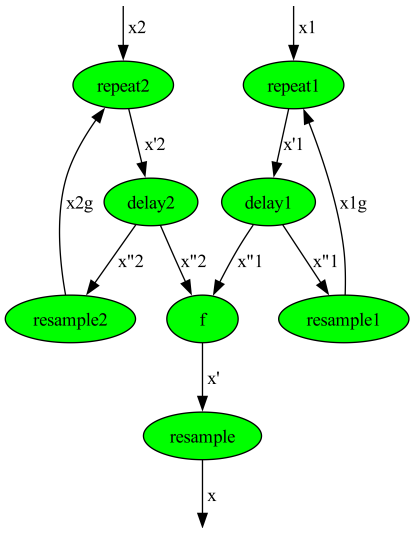
$$\frac{\begin{array}{c} e_d \mapsto \mathsf{V}_d; x_d \\ e_c \mapsto \mathsf{V}_c; x_c \\ l = \text{free}(e_p) \\ \theta = \{y \to y' \,|\, y \in l\} \cup \{\mathbf{r} \to \mathbf{r}'\} \quad y' \text{ fresh} \quad \mathbf{r}' \text{ fresh} \\ x \text{ fresh} \quad x_b \text{ fresh} \\ v_{deadline} = \langle [x, x_b] = \texttt{test}(x_d, x_c), \texttt{TBT}, \langle\langle -\infty, [-\infty, -\infty]\rangle\rangle\rangle \\ v_{planb} = \langle (\text{list}(l) + [\mathbf{r}]) \circ \theta = \texttt{branch}(x_b, \text{list}(l)), \texttt{DV}, \cdot\rangle \\ e_p \mapsto \mathsf{V}_p; x_p \\ v_{out} = \langle [x_{out}] = \texttt{merge}(x, x_p), \texttt{Immediate}, \cdot\rangle \quad x_{out} \text{ fresh} \end{array}}{\begin{array}{c} \texttt{TTFinishByOtherwise}(e_d, \texttt{TTTimeDeadline=}e_c, \\ \texttt{TTPlanB=}e_p, \texttt{TTWillContinue=True}) \mapsto \\ \mathsf{V}_d \cup \mathsf{V}_c \cup (\mathsf{V}_p \circ \theta) \cup \{v_{deadline}, v_{planb}, v_{out}\}; x_{out} \end{array}} \textit{ Deadline Catch} \tag{2.6}$$

The `TTSingleRunTimeout` ensures that only one execution of e can occur in the graph. It does so by using the `DV` firing rule and introducing a synthetic port x_m. The token in x_m functions very similarly to mutual exclusion, since it must be consumed before e can run. x_m allows v_{gather} to wait until its prior execution has finished before allowing further execution. When the graph is instantiated, a token with a **`True`** data value is populated for x_m in this `branch` SQ. By construction, either a token is in the *waiting-matching section* for x_i or it has been consumed and the subgraph is firing. If e is executing, no other executions can happen because they lack a token in x_m. The SQs v_{delay} and $v_{resample}$ are used to exclude synchronizations for the new token for x_m by purposefully offsetting its time interval into the future. The delay begins after e to ensure that any tokens generated during e's execution is ignored. This prevents it from matching with tokens generated before it was generated.

When `delay` runs, it takes the current time t and either waits until `first_offset` time steps later if specified and has not been run before, otherwise waits `offset` time steps later. The function `resample` takes its input token and reassigns its time interval. It takes the current time t and set the token's new time interval as $[t - \texttt{start\_offset}, t + \texttt{end\_offset}]$. We set `end_offset` to be ∞ to match with any future tokens.



(a) Graph compilation of Rule Single Run where the enclosed expression uses the free variable a. v_{gather} is represented by the **SINGLETON** node. Both x_m's point to the same port.



(b) Graph compilation of Rule `STREAMify` for the function `def f`(x_1, x_2). Each cycle is responsible for periodically creating one of the function's parameters.

Figure 2.5: Compilation graphs for *Single Run* and `STREAMify`.

$$\frac{\begin{array}{c} l = \text{free}(e) \quad x_m \text{ fresh} \\ \theta = \{y \to y' \,|\, y \in l\} \cup \{\mathbf{r} \to \mathbf{r}'\} \quad y' \text{ fresh} \quad \mathbf{r}' \text{ fresh} \\ v_{gather} = \langle (\text{list}(l) + [\mathbf{r}]) \circ \theta = \texttt{branch}(x_m, \text{list}(l)), \texttt{DV}, \cdot \rangle \\ e \mapsto \mathsf{V}; x_o \\ v_{delay} = \langle [x'] = \texttt{delay}(x_o, \text{offset} = n), \texttt{DV}, \cdot \rangle \\ v_{resample} = \langle [x_m] = \texttt{resample}(x', \texttt{start\_offset} = 0, \\ \qquad \texttt{end\_offset} = \infty), \texttt{DV}, \cdot \rangle \end{array}}{\texttt{TTSingleRunTimeout}\,(e, \texttt{TTTimeout=}n) \mapsto \{v_{gather}, v_{delay}\} \cup (\mathsf{V} \circ \theta); x_o}\ \textit{Single Run} \tag{2.7}$$

A function decorated with `@SQify` compiles to a SQ encapsulating that function.

$$\frac{\forall i = 1..n.\ e_i \mapsto \mathsf{V}_i; x_i \quad x \text{ fresh}}{\begin{array}{c} f(e_1, \cdots, e_n) \mapsto \\ \bigcup_{\forall i = 1..n.} \mathsf{V}_i \cup \{\langle [x] = f(x_1, \cdots, x_n), \texttt{DV}, \cdot \rangle\}; x \end{array}}\ \texttt{SQify}\ \textit{Func} \tag{2.8}$$

A function decorated with `@STREAMify` compiles to a subgraph of SQs to describe periodic capabilities. A STREAMified function produces tokens periodically from one set of synchronized inputs. It will produce tokens with the specified period and phase until the current time no longer synchronizes with the original synchronized inputs. This allows us to specify when periodic functions start and end. As an execution consumes the input tokens, the graph needs to regenerate the input tokens repeatedly. The SQs v_i, v_i', and v_i^r are responsible for generating new tokens for the next iteration and timestamping them accordingly. v_i is responsible for propagating the first set of tokens to the STREAMified function f and

the regenerated ones over time. Its function `repeat` behaves differently if it gets tokens from x_i (the start of the periodic call) or x_i^g (generated from a prior iteration). The latest timestamped token k_s received on x_i is saved in the SQ state s and then sent to the STREAMified function in v_f. When tokens come in from x_i^g, they are compared to k_s. If they are not synchronized with k_s, they are not propagated to v_f. v_i' and v_i^r create and timestamp tokens for the next iteration. v_i''s function `delay` has a new parameter `first_offset` where its first execution will delay by n_2. All subsequent executions will delay by n_1.

$$\frac{\begin{array}{c}\forall i = 1..n.\ e_i \mapsto \mathsf{V}_i; x_i \\ x_i' \text{ fresh} \quad x_i^g \text{ fresh} \quad x_i'' \text{ fresh} \\ v_i = \langle [x_i'] = \texttt{repeat}(x_i, x_i^g), \texttt{Immediate}, \langle -\infty, [-\infty, -\infty] \rangle \rangle \\ v_i' = \langle [x_i''] = \texttt{delay}(x_i', \texttt{first\_offset} = n_2, \texttt{offset} = n_1), \texttt{DV}, \cdot \rangle \\ v_f = \langle [x'] = f(x_1'', \cdots, x_n''), \texttt{DV}, \cdot \rangle \quad x' \text{ fresh} \\ v_i^r = \langle [x_i^g] = \texttt{resample}(x_i'', \texttt{start\_offset} = -n_3/2, \\ \qquad \texttt{end\_offset} = n_3/2), \texttt{DV}, \cdot \rangle \\ v_r = \langle [x] = \texttt{resample}(x', \texttt{start\_offset} = -n_3/2, \\ \qquad \texttt{end\_offset} = n_3/2), \texttt{DV}, \cdot \rangle \quad x \text{ fresh}\end{array}}{\begin{array}{c} f(e_1, \cdots, e_n, \texttt{TTPeriod=}n_1, \\ \texttt{TTPhase=}n_2, \texttt{TTDataIntervalWidth=}n_3) \mapsto \\ (\bigcup_{\forall i=1..n.} \mathsf{V}_i \cup \{v_i, v_i', v_i^r\}) \cup \{v_f, v_r\}; x \end{array}}\ \texttt{STREAMify}\ \textit{Func} \tag{2.9}$$

Compiling Statements

Statements in a GRAPHified function provide variable names for expressions.

$$\frac{e \mapsto \mathsf{V}; x \quad \theta = \{(x \to y)\}}{y = e \mapsto \mathsf{V} \circ \theta; y}\ \textit{Assign}$$

Statements also allow for code blocks that provide runtime information such as mapping. **`with`** statements provide runtime information through `TTClock` and `TTConstraint` constructs. These statements add labels for the subgraph of SQs generated in its block. TTPython uses these labels during the deployment of these SQs. We have not added these labels to the definition of SQs for brevity of compilation and semantic rules.

2.4 Timed, Tagged-Token Dataflow Semantics

We first describe the system configuration $\mathcal{C}$ of the DFG. This is described as the tuple

$$\mathcal{C} = \langle \mathbb{G}, \mathcal{N}, t \rangle$$

where $\mathbb{G}$ is the dataflow graph, $\mathcal{N}$ is a set to hold onto tokens in transit through the network, and t is the current time for the system. An element of $\mathcal{N}$ is described as a tuple $\langle k, \mathtt{x} \rangle$ where k is a token and $\mathtt{x}$ is the name of the input port it is addressed to.

Sometimes, there are no possible actions for $\mathbb{G}$ because there are no fulfillable firing rules or tokens are still traveling through the network $\mathcal{N}$. To simulate this, a rule describes the passage of time with no actions from the dataflow graph or network. We represent this with the following rule:[3]

[3]This rule has lower precedence than the future rules described later in this section.

$$\frac{}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G}, \mathcal{N}, t+1\rangle}\ \textit{Inc-Time} \tag{2.10}$$

Tokens can also exit from the network. They are added to the waiting-matching set of their destination input port.

$$\frac{\begin{array}{c}\langle k, \mathtt{x}\rangle \in \mathcal{N} \quad \mathsf{V} = \{v \,|\, v \in \mathbb{G} \wedge i \in v.\bar{i} \wedge \mathtt{x} = i.\mathtt{x}\} \\ \mathsf{V}' = \{v[i \rightarrow \langle x, i, i.\mathtt{W} \cup \{k\}\rangle] \,|\, v \in \mathsf{V}\}\end{array}}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G} \setminus \mathsf{V} \cup \mathsf{V}', \mathcal{N} \setminus \{\langle k, \mathtt{x}\rangle\}, t\rangle}\ \textit{Rcv-Tok} \tag{2.11}$$

The notation for Line 2 in Rule Rcv-Tok is shorthand for updating the SQ's input port with the new waiting-matching set unioned with the token to be added. To start a nontrivial execution, an initial configuration begins with a nonempty $\mathcal{N}$ with tokens with that have r as their input port destination. The system generates new tokens by consuming tokens from its input ports and sending them to the network. A SQ's firing rule describes this behavior.

2.4.1 Firing Rule: Immediate

The immediate firing checks if a token is in the waiting-matching section of any of its firing ports. If so, it will fire immediately while providing null values (`None` in Python) for all other input ports.

$$\frac{\begin{array}{c} v = \langle \overline{o} = f(\bar{i}), \mathtt{DV}, s\rangle \in \mathbb{G} \wedge \mathtt{len}(\bar{i}) = n \wedge \bar{i} = [i_1, \cdots, i_n] \\ \exists j.\, 1 \leq j \leq n \wedge k \in i_j.\mathtt{W} \\ f(\mathtt{None}, \cdots, k, \cdots, \mathtt{None}) =^* \mathcal{N}' \\ \mathtt{W}' = i.\mathtt{W} \setminus \{k\} \\ v' = \langle \overline{o} = f(\langle i_1, \cdots, \langle i_j.\mathtt{x}, \mathtt{W}'\rangle, \cdots, i_n\rangle), \mathtt{DV}, s\rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t\rangle}\ \textit{I-FR} \tag{2.12}$$

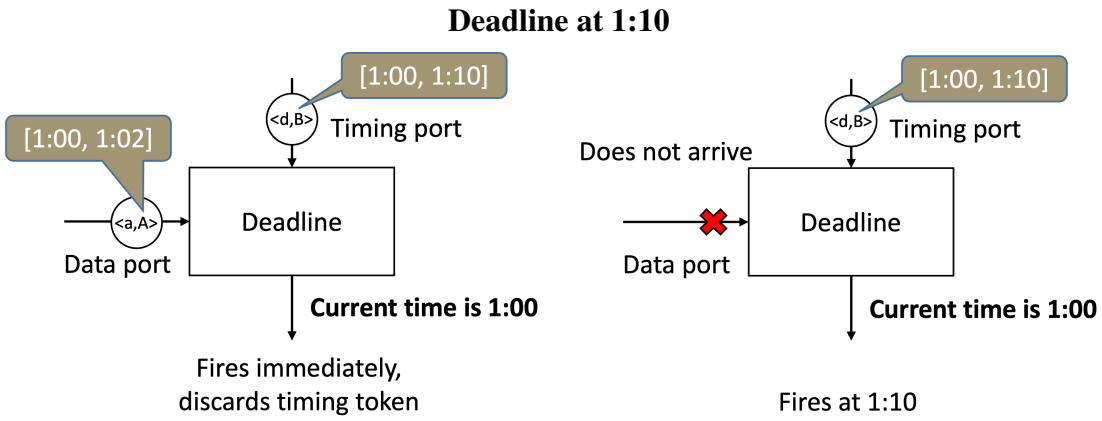
2.4.2 Firing Rule: Data-Validity

A SQ v with the data-validity firing rule will fire if there exists a token on each input port of v such that their time intervals overlap. Tokens $k_i = \langle d_i, [t_s^i, t_e^i]\rangle$ and $k_j = \langle d_j, [t_s^j, t_e^j]\rangle$ have overlapping time intervals ($k_i \cap k_j \neq \emptyset$) if $t_s^i \leq t_e^j$ and $t_s^j \leq t_e^i$. The firing rule is described by the rule below:

$$\frac{\begin{array}{c} v = \langle \overline{o} = f(\bar{i}), \mathtt{DV}, s\rangle \in \mathbb{G} \wedge \mathtt{len}(\bar{i}) = n \wedge \bar{i} = [i_1, \cdots, i_n] \\ \exists\, k_1, \cdots, k_n.\, \forall\, a, b \in 1..n. \\ a \neq b \implies k_a \in i_a.\mathtt{W} \wedge k_b \in i_b.\mathtt{W} \wedge k_a \cap k_b \neq \emptyset \\ f(k_1, ..., k_n) =^* \mathcal{N}' \\ \forall j \in 1..n.\ \mathtt{W}_j{}' = i_j.\mathtt{W} \setminus \{k_j\} \\ v' = \langle \overline{o} = f(\langle i_1.\mathtt{x}, \mathtt{W}_1{}'\rangle, \cdots, \langle i_n.\mathtt{x}, \mathtt{W}_n{}'\rangle), \mathtt{DV}, s\rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t\rangle}\ \textit{DV-FR} \tag{2.13}$$

2.4.3 Firing Rule: Time-Based Trigger

A SQ with this firing rule has two input ports, delineated as i_d and i_c. i_d corresponds to a data port, and i_c is the control port. We label tokens k_d and k_c for the i_d and i_c input ports respectively. The list of output ports $\overline{o}$ split computation into two branches: downstream SQs that use the data token and Plan B's



(a) Normal Execution (b) Plan B Execution

Figure 2.6: Time-Based Trigger Example Firing Semantics

associated subgraph, so we simplify $\overline{o}$ to two output ports $[o_d, o_c]$ to perform normal or Plan B execution respectively. The SQ acts as a pass-through when the deadline time is satisfactorily met.

The SQ will internally keep track of the last control token it has seen from i_c that caused Plan B to run, which we denote as k'_c. This is tracked in $v.s$, the SQ's internal state. k_c encodes the deadline in its time interval. $k_c.t_s$ denotes when the deadline was initiated and $k_c.t_e$ is the timestamp of the deadline.

The semantic rules guarantee that Plan B fires at most once per control token. It will either fire if the data token does not arrive on time or discard the control token otherwise. We first describe the success case. Summarily, success is when both conditions are satisfied:

1. Tokens arrive on time (*i.e.*, they arrive before the deadline specified by the control token).

2. Data is synchronized with the control token (*i.e.*, they overlap).

$$\frac{\begin{array}{c} v = \langle [o_d, o_c] = \texttt{test}(i_d, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_d \in i_d.\texttt{W} \quad k_c \in i_c.\texttt{W} \quad t \leq k_c.t_e \quad k_d \cap k_c \neq \emptyset \\ \bar{i} = [\langle i_d.\texttt{x}, i_d.\texttt{W} \setminus \{k_d\} \rangle, \langle i_c.\texttt{x}, i_c.\texttt{W} \setminus \{k_c\} \rangle] \\ v' = \langle [o_d, o_c] = \texttt{test}(\bar{i}), \texttt{TBT}, \langle k'_c \rangle \rangle \\ \mathcal{N}' = \{\langle k_d, x \rangle \mid x \in o_d\} \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t \rangle} \; \textit{TBT-S} \tag{2.14}$$

Exceptional cases occur when either condition is not met.

1. Data is late (*i.e.*, time t is greater than the control token k_c's end timestamp).

   Plan B is run (Rule TBT-F(ailure)-P(lan)BC(ontrol)), and we will discard any data tokens that would be synchronized to this control token (Rule TBT-F(ailure)-P(lan)BD(ata)). The function $\texttt{max}(k, k')$ returns the token with the greater t_e.

$$\frac{\begin{array}{c} v = \langle [o_d, o_c] = \texttt{test}(i_d, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_c \in i_c.\mathbb{W} \quad k_c.t_e < t \\ i_c^n = \langle i_c.\texttt{x}, i_c.\mathbb{W} \setminus \{k_c\} \rangle \\ v' = \langle [o_d, o_c] = \texttt{test}(i_d, i_c^n), \texttt{TBT}, \langle \max(k_c, k'_c) \rangle \rangle \\ \mathcal{N}' = \{\langle k_c, x \rangle \mid x \in o_c\} \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t \rangle} \textit{ TBT-F-PBC} \tag{2.15}$$

$$\frac{\begin{array}{c} v = \langle [o_d, o_c] = \texttt{test}(i_d, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_d \in i_d.\mathbb{W} \quad k_d.t_s \leq k'_c.t_e \\ i_d^n = \langle i_d.\texttt{x}, i_d.\mathbb{W} \setminus \{k_d\} \rangle \\ v' = \langle [o_d, o_c] = \texttt{test}(i_d^n, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N}, t \rangle} \textit{ TBT-F-PBD} \tag{2.16}$$

2. Data has arrived but is unsynchronized.

   This failed condition has three cases:

   - **Data token has arrived but there is no overlapping control token.**

     The SQ will wait for a control token to come.

   - **Control token arrives after its specified deadline.**

     As the deadline has already passed, the SQ must run Plan B immediately. This is already encoded with Rule TBT-F-PBC.

   - **Control and Data token do not match.** Depending on the reliability of the network, some tokens may never arrive at their destination. This raises concerns of memory impact, as unsynchronized tokens will wait forever in the waiting-matching section without a policy to evict them. The rule below addresses how to dispose of data tokens if their corresponding control tokens never arrive for the `TBT`firing rule. The `DV`firing rule for *cleaning* unsynchronized data tokens follows a similar format of using a timeout. The TTDA graph literature describes this phenomenon as *self-cleaning*.

     This rule depends on the order of the time intervals between the control and data tokens. We assume here that $k_d \cap k_c = \emptyset$.

     If k_d's time interval is before k_c, the SQ will wait until time is greater than $k_d.t_e + \texttt{Exp}$, where `Exp` is a programmer-defined time extension to wait for the control token. `Exp` by default is upper-bounded by the periodicity of the stream that k_d has been generated by. The default ensures in-order operation for periodic streams. This implies that if iteration n of the control token in the stream arrives, any tokens from iteration $n-1$ are automatically late (unless if `Exp` is set explicitly longer by the programmer).

$$\frac{\begin{array}{c} v = \langle [o_d, o_c] = \texttt{test}(i_d, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_d \in i_d.\mathbb{W} \quad k_c \in i_c.\mathbb{W} \\ k_d.t_e < k_c.t_s \quad k_d.t_e + \texttt{Exp} < t \\ i_d^n = \langle i_d.\texttt{x}, i_d.\mathbb{W} \setminus \{k_d\} \rangle \\ v' = \langle [o_d, o_c] = \texttt{test}(i_d^n, i_c), \texttt{TBT}, \langle k'_c \rangle \rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N}, t \rangle} \textit{ TBT-F-US} \tag{2.17}$$

If k_c's time interval is before k_d, the SQ waits for the corresponding data and control tokens for each as the data for the next iteration has arrived earlier than expected.

2.5 Running an Application in TTPython

The programmer first compiles a TTPython program into a dataflow graph. The TTPython architecture handles many runtime considerations, such as SQ setup and distribution, SQ firing, code execution, and forwarding output to downstream SQs. The TTPython compiler structures SQs in three components to aid the runtime. Each SQ has three components: Synchronization, Computation, and Communication, as shown in Figure 2.7. Our abstractions manifest in Synchronization and Communication; the Computation section is simply the user's code. Synchronization implements the waiting-matching section and enables Computation once the SQ's firing rule is satisfied. Computation contains the decorated functions specified by the programmer with `@SQify` or `@STREAMify` and runs to completion. We use the `pathos` framework [42, 43] to avoid Python's GIL to run Plan B SQs in a timely manner. Communication encapsulates data values into tokens with a tag before sending them to downstream SQ(s) and, if necessary, over the network.

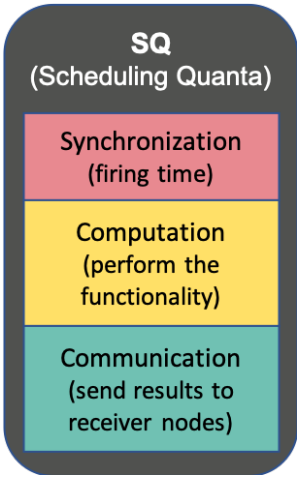


Figure 2.7: The components of a Scheduling Quantum

To execute the program, the encoded DFG is provided to a Runtime Manager (RTM), *i.e.*, a privileged device with knowledge of available devices in the running application. The RTM maps the SQs constituting the program onto those available devices based on program SQ constraints. Instantiating the program across the devices in the system first requires knowledge of those devices. TTPython first holds a joining phase, in which devices contact the RTM and inform it of their capabilities. This includes a unique name, an IP address, hardware and software components, and a port through which the device will accept messages. The RTM uses this to produce a dictionary linking names to network addresses. Before the program starts, the RTM distributes this table among the online devices so they can communicate directly with each other and run the program in a decentralized manner. Propagating the routing table across devices allows them to run asynchronously without the RTM's support. Once the RTM maps SQs to devices, the SQs are sent over the network to the devices running those corresponding SQs. Each device will unpack these SQs and await incoming tokens. When all devices acknowledge the SQs they will run, the program is ready to begin. The RTM then creates and sends starter tokens to the SQs that take externally sourced tokens. SQs that operate on no inputs (such as a stream generator or constant SQ) will receive a dummy token. The dataflow graph begins executing and will continue until no tokens are left and no stream generators remain active.

Each device that is running TTPython has three processes responsible for handling the components of a SQ: Synchronization, Computation, and Communication.

Chapter 3

Case Studies

TTPython has been used to implement two practical DT applications: the smart intersection (SI) and the urban flooding (UF) network. The smart intersection concretizes code examples that show how TTPython abstracts many implementation details necessary for DT systems. The urban flooding network generalizes TTPython's expressiveness in a different setting and shows how hardware considerations affect DT application design. We motivate each application before discussing results in detail.

3.1 Smart Intersection

Cooperative autonomous vehicle intersections present a unique problem in terms of distribution and timing. The smart intersection uses roadside sensors with connected autonomous vehicles to cooperatively coordinate vehicle planning and movement through the intersection. The goal of an autonomous vehicle intersection is to allow the Connected Autonomous Vehicles (CAVs) to drive through the intersection as close to the speed limit as possible without colliding [32]. Vehicles traveling at high velocity near each other have little margin for error. Accomplishing this feat requires precise timing, sensor fusion (the interpretation of the environment state from different sensors), communication, and execution of the planned path. Any deviation or failure could be catastrophic.

The smart intersection application consists of a multitude of sensor streams that must be combined into a single worldview (global fusion), which is then used to determine the paths each CAV will follow in the physical world. A connected infrastructure sensor (CIS) assists by providing a stationary camera that provides localized information at the intersection. A Road Side Unit (RSU) takes the information from the incoming CAVs and CIS to determine the paths each CAV should take. Figure 3.1 depicts a combined CAV and CIS dataflow diagram. All processes of the CIS are depicted in blue while those of the CAV are in blue and gray. A CAV has an extra LIDAR sensor and can actuate a steering and drive motor to move, whereas the CIS sensor is just for sensing alone and has no actuation ability. All CAVs and CIS sensors in the network must synchronize their sensor frequency within a tight margin so that sensor fusion operates correctly. All CAVs and CISs must process and locally fuse their data to send to the RSU so that the RSU has enough time to calculate the global fusion and intersection control [31] (see Figure 3.2). The data is then sent out to all the CAVs so they can actuate their steering and motors. This entire process must happen within 0.125ms so that timing errors do not cause a crash of the CAVs. However, timing and communication are very sensitive to problems. Even slight timing issues can cause the perceived positions and trajectories of the CAVs to be estimated incorrectly, which can result in a crash as the CAV is actually at a different position. Additionally, any late or out-of-order communications can result in a crash because

these errors may also affect the perceived positions.

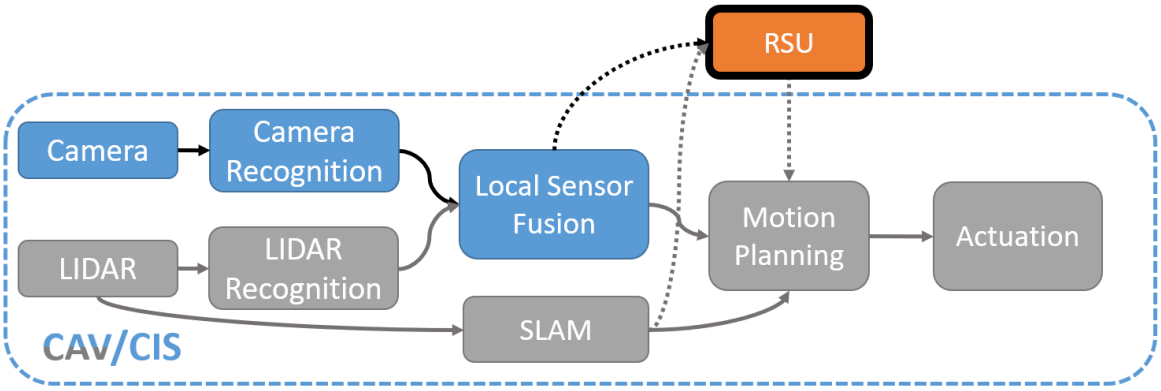


Figure 3.1: Dataflow of a Connected Autonomous Vehicle (CAV) shown in blue and gray and a Connected Infrastructure Sensor (CIS) shown in blue. CAVs and CISs send their locally fused sensor data to the RSU and CAVs receive intersection control back which is used to actuate the steering and motors.

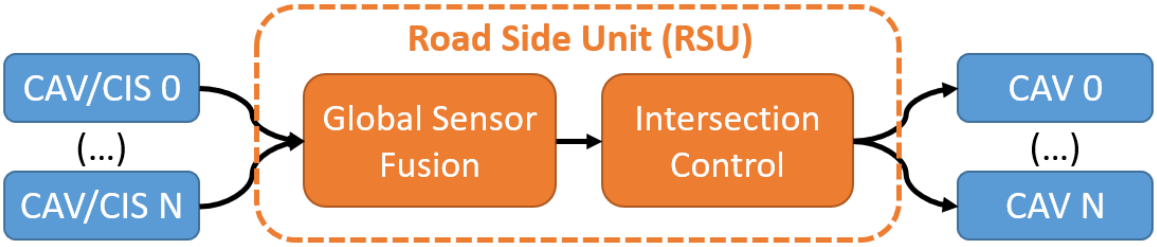


Figure 3.2: Dataflow of a Road Side Unit (RSU) that gathers sensing data from the CAVs and CISs in the area, processes the global sensor fusion, and calculates the intersection controls to send back out to the CAVs in the area.

This case study application was created using 1/10-scale autonomous vehicle models with scale-accurate LIDAR and camera sensors shown in Figure 3.3. These work with a 1/10-scale connected infrastructure sensor (CIS), which is a stationary camera that also shares information with the intersection, as shown in Figure 3.4. The 1/10-scale vehicles drive a figure eight loop with an intersection in the middle, which can be seen in Figure 3.5. This intersection is controlled using an autonomous vehicle intersection controller running on a Jetson TX2. The intersection has four scale CAVs to control, 2 scale CISs, and a roadside unit (RSU) running the intersection controller.

Our research questions for this case study are as follows.

RQ 1 **How do TTPython's abstractions shorten infrastructure code in DT applications?**

RQ 2 **How do TTPython's abstractions expose bugs in infrastructure code in DT applications?**

3.1.1 Methodology

We took a completed 1/10-scale CAV application and rewrote it using TTPython. The developer (an experienced DT application programmer) of the application was responsible for the conversion. We then

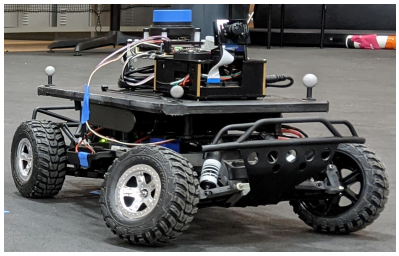


Figure 3.3: One tenth scale CAV with camera, LIDAR, and Nvidia Jetson Nano for on-board processing.

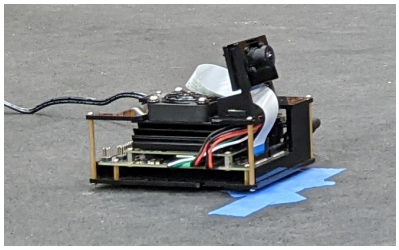


Figure 3.4: One tenth scale CIS with camera using Nvidia Jetson Nano for on-board processing.

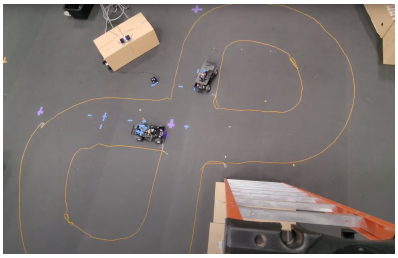


Figure 3.5: Overhead of one tenth scale CAVs shown driving within the figure 8 intersection.

performed a comparison of the code implementation and execution in the field. We highlight specific code from the original application abstracted by TTPython's timing and distribution constructs. The two implementations were deployed on two CAVs and an RSU, and we observed execution behavior. We collected various timing data, such as how long a periodic iteration ran before looping and how long it took for the motor to actuate periodically.

3.1.2 Code Comparison

We compared the original code for the smart intersection application to a newly converted TTPython version. To do so, we manually annotated the application code for both applications with labels that involved infrastructure code. These were categorized into timing, concurrency, and networking. We excluded extra newlines introduced by formatting, and then counted the labels. The original application has 6415 lines of Python code, 624 of which is infrastructure code, which consists of code responsible for timing, concurrency, and networking. The functionality of both code bases is identical from the application output perspective. However, many of the internal structures were changed when migrating the original Python code base to TTPython. These changes fall into three main categories: 1) timing management code ensuring that sensors and devices sense and actuate at the proper time, 2) concurrency, including pipelines for inter-process communication, and 3) networking code allowing all the different devices to communicate. TTPython reduces the code needed for all three of these categories and eliminates 95% of the code in these three categories combined, as seen in Table 3.1.

(LoC)	Original App	TTPython Variant
Timing	255	23
Concurrency	29	0
Networking	340	5

Table 3.1: Breakdown of infrastructure lines of code. TTPython eliminates concurrent and networking code with the DFG's graph abstraction and dynamic network construction.

Timing Management Code is Drastically Reduced

With regards to RQ 1, we see that TTPython localizes the timing requirements of the code when creating specific timed control flow constructs. SQ firing rules ensure that the cameras and LI-

DARs stay synchronized at the desired frequency and within a certain time interval. Having the programmer specify timing before execution clarifies periodic and recovery mechanisms with Plan B. This direct support for timing management improved the brevity of the code from 255 lines of Python code to 23 lines of TTPython. For example, in TTPython a few lines of code such as `cam_sample = camera_sampler(cav_0, sample_window, TTClock=root_clock, TTPeriod=125000, TTPhase=0, TTDataIntervalWidth=62000)` replaced 30 lines of manually written Python code to manage the timing and synchronization directly. These abstractions enabled TTPython to reduce timing code by 90% while making the code more declarative and easier to understand.

Listing 3.1: The original application's implementation of a CAV using Python's time and multiprocessing library with a user written API over a 3rd-party communication library. The timing, distribution, and concurrency code is highlighted respectively in yellow, blue, and turquoise.

```
# Start after 10 seconds
start_time = time.time() + 10_000_000
interval = 125_000 # Interval is 125ms

# Sleep until test start time
wait_until_start = (
    start_time - time.time() -.01
)
if wait_until_start > 0:
    time.sleep(wait_until_start)

# Spawn the camera processing process
cam_out_queue = Queue()
cameraProcess = Process(target=sourceImagesProcess,
    args=(cam_out_queue, settings, camSpecs,
    simulation_time, data_collect_mode,
    start_time, interval))
cameraProcess.start()

# Spawn the lidar processing process
lidar_out_queue = Queue()
lidarProcess = Process(target=sourceLIDARProcess,
    args=(lidar_out_queue, pipeFromC, pipeToC,
    planner.lidarSensor, simulation_time,
    data_collect_mode, start_time, interval))
lidarProcess.start()

# Spawn the communication process
manager = Manager()
init = manager.dict()
response = manager.dict()
response["error"] = 1
# process initialization omitted, similar 3 lines
# as above two process initializations

# Sleep process until target time
target = start_time
while True:
  if target <= time.time():
    now = time.time()
    lidar_received = False
    camera_received = False

    fallthrough = now + fallthrough_delay
    while time.time() < fallthrough and not
            (lidar_received and camera_received):
        # Get the lidar
        if not lidar_out_queue.empty():
```

```
49              lidar_returned = lidar_out_queue.get()
50              lidar_received = True
51          # Get the camera
52          if not cam_out_queue.empty():
53              cam_returned = cam_out_queue.get()
54              camera_received = True
55
56      if lidar_received and camera_received:
57          # normal execution
58      else:
59          deadline_check()
60
61      # Prep value to be sent, clear queue of old data
62      while not out_queue.empty():
63          out_queue.get()
64          out_queue.put(
65              [camcoordinates,
66              now,
67              time.time()]
68          )]
69      target = target + interval
70    time.sleep(.001)
```

Listing 3.2: The SI application rewritten using TTPython. Listing 3.1's code is comparable to Lines 1-27

```
1  ##### Comparable section to 3.1
2  N = 1_000
3  cav_0 = 0
4  # starts N seconds later
5  start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 1_000_000 * N
6  stop_time = GET_INFINITY(trigger, TTClock=root_clock)
7  sampling_time = VALUES_TO_TTTIME(start_time, stop_time)
8  cav_0_2 = COPY_TTTIME(cav_0, sampling_time)
9
10 with TTConstraint(name="cav0"):
11     cam_processed, cam_sample = camera_process(cav_0_2,
12         TTClock=root_clock,
13         TTPeriod=250_000,
14         TTPhase=0,
15         TTDataIntervalWidth=500_000
16         TTFirstInstanceDelay=10_000_000,
17         TTPersistent=True)
18     lidar_sample = lidar_process(cav_0_2,
19         TTClock=root_clock,
20         TTPeriod=250_000,
21         TTPhase=0,
22         TTDataIntervalWidth=500_000
23         TTFirstInstanceDelay=10_000_000,
24         TTPersistent=True)
25     fusion_result0 = local_fusion(cam_processed, lidar_processed,
26                                   cav_0_2, TTPersistent=True)
27 ##### Comparable section to 3.1
28
29 with TTConstraint(name="rsu"):
30     rsu_deadlines, _ = rsu_deadline_dummy(
31         sample_window,
32         TTClock=root_clock,
33         TTPeriod=250_000,
34         TTPhase=0,
35         TTDataIntervalWidth=550_000,
36         TTFirstInstanceDelay=10_000_000,
37         TTPersistent=True)
38     global_fusion_input_timeout = READ_TTCLOCK(rsu_deadlines, TTClock=root_clock) + 500_000
39     fusion_result_deadline_rsu = TTFinishByOtherwise(
40         fusion_result,
```

```
41          TTTimeDeadline=global_fusion_input_timeout,
42          TTPlanB=missed_fusion_input(),
43          TTWillContinue=True)
44      global_fusion_result_rsu = global_fusion(
45          fusion_result_deadline_rsu)
46
47  with TTConstraint(name="cav0"):
48      local_fusion_timeout_cav0 = READ_TTCLOCK(cam_sample, TTClock=root_clock, delay=550_000)
49      local_planner_timeout_cav0 = READ_TTCLOCK(cam_sample, TTClock=root_clock, delay=600_000)
50      global_fusion_result_deadline_cav0 = TTFinishByOtherwise(
51          global_fusion_result_rsu,
52          TTTimeDeadline=local_fusion_timeout_cav0,
53          TTPlanB=no_global_fusion(),
54          TTWillContinue=True)
55      lidar_processed_deadline_cav0 = TTFinishByOtherwise(
56          lidar_processed,
57          TTTimeDeadline=local_planner_timeout_cav0,
58          TTPlanB=no_lidar(),
59          TTWillContinue=True)
60      command_velocity_cav0 = calculate_angle(
61          global_fusion_result_deadline_cav0,
62          lidar_processed_deadline_cav0, cav_0_2)
63      final_result = command_motors(command_velocity_cav0)
```

Examining Listing 3.1, the original CAV application made direct use of the Python time library to implement waiting for the right time to start computation and a **while** loop to implement periodicity with a `time.sleep()` to run every millisecond. It checks the validity of the data by keeping boolean flags to track its synchronization between the LIDAR and camera through `lidar_received` and `camera_received`. Its comparable implementaion is seen in Listing 3.2. TTPython separates these concerns from the programmer into 4 steps: setting a start time, stop time, periodicity, and data interval validity for downstream tokens. This code is built into the TTPython runtime system and thus does not have to be written by the applications programmer; eliminating it is one of the reasons for the reduction in code size mentioned above.

Concurrency is Eliminated and Networking Management Code is Reduced

We observe that the inherent concurrent behavior of dataflow graphs eliminates the overhead of concurrency and networking management. The DFG by construction creates data race free execution between its SQs. The programmer writes functional aspects of the code combined with local state. State sharing is managed through function calls that translate to message passing, so there is no need to write process management code. This completely eliminates 29 lines of process creation and management from the original application. Communication is also built into the TTPython dataflow architecture. The original implementation used Flask, a Python web framework. Although it was a popular choice for networking, it was not trivial to implement the interactions between the CAVs and RSU as discussed in Section 1.1. Additionally, timing data structures, fallback routines, and routing tables had to be created manually, whereas TTPython generates those automatically. TTPython thus reduced 340 lines of code to 5 with the `TTConstraint` feature.

Referring back to Listing 3.1, there are explicit calls to the network queues used for connections to other devices. This is seen in the `out_queue` variable after the function call `camera_sampler`. In the original implementation, each device required a separate file responsible for handling communication. TTPython avoids this by abstracting communication between devices through the dataflow graph. The dataflow graph uses variables to name input and output ports for SQs. As SQs are the smallest unit of execution, once the runtime manager maps and assigns all SQ locations, it can generate a routing table for each device to specify where to send the output tokens per SQ. On Lines 25 and 44, `fusion_result0` is

sent to the RSU for its `global_fusion` function with no networking or synchronization code necessary from the programmer side. This abstraction also makes it easy for the programmer to easily move code across devices. The programmer simply needs to move code across **`with`** blocks. Under the hood, TTPython modifies the routing tables during its mapping phase but does not need to touch the functional aspect of the SQ. This further simplifies the project structure as shown in Figure 3.6. In the original application, there are separate folders for the CAV and RSU code. Any shared communication would first need to interface with the networking library before other devices could receive it. Sharing is simple in TTPython, as **`with`** blocks with `TTConstraint` do not limit variable scoping. Sharing variable names reduces the complexity of interfacing with different web frameworks and identifying properties and invariants of the framework for correct execution. TTPython reduces programmer work concerning concurrent and networking code.

3.1.3 Execution Analysis

To help answer RQ 2, we have run both TTPython and the vanilla Python implementation of the smart intersection to identify runtime errors in the original implementation. Two CAVs run connected to a shared RSU. Each implementation is run 3 times to obtain the averages for each of the following table data samples. We first allowed each implementation to reach a steady state before tracking data over a 10-minute run. We wrote timing information to a file to track how long an iteration takes and how often code executes between periodic executions. The timestamps from each CAV's timing information file were then aligned for comparison.

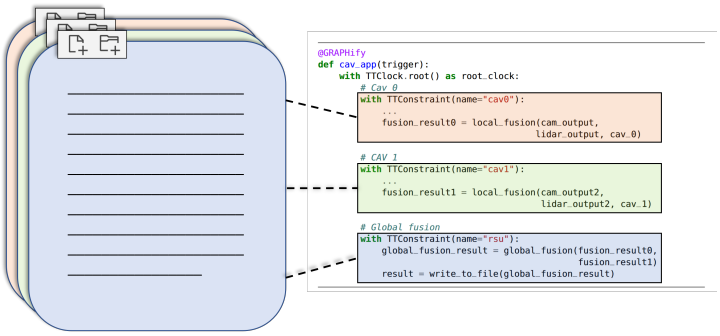


Figure 3.6: Showcasing macroprogramming's system-level view. The original implementation had separate files designated for each device, while TTPython supports development of all constituent devices seamlessly in one file.

We noticed that running the application in TTPython was noticeably slower than the original application. The original application would complete a periodic cycle on the CAV within 125ms, while the TTPython version would take around 250ms. We initially assumed that this was caused by TTPython's overhead, as TTPython adds extra data synchronization checks between each SQs. This would add data synchronization between the asynchronous data generators (camera and LIDAR on the CAVs) and when `global_fusion` runs with local fusion data from both of the CAVs. However, on closer inspection of timing information and the networking code, we identified two bugs associated with the network in Listing 4.

```
1 def processCommunications(comm_q, v_id, init, response, rsu_ip):
2     # initialization code
3     ...
4
5     while 1:
6         if not comm_q.empty():
```

```
7               got = comm_q.get()
8               response_message = rsu.checkin(*got)
9
10              if response_message == None:
11                  response["error"] = 1
12              else:
13                  response["error"] = 0
14                  response["v_t"] = response_message["v_t"]
15                  response["tfl_state"] = response_message["tfl_state"]
16                  response["veh_locations"] = response_message["veh_locations"]
17
18  def cav(config, vid):
19      # Spawn the communication thread
20      manager = Manager()
21      init = manager.dict()
22      response = manager.dict()
23      response["error"] = 1
24      comm_q = Queue()
25      commProcess = Process(target=processCommunicationsThread,
26                            args=(comm_q, vehicle_id, init,
27                                  response, config.rsu_ip))
28      commProcess.start()
29
30      while True:
31          # compute local fusion and send it over the network
32          ...
33          comm_q.put(local_fusion_data)
34          time.sleep(.01)
35
36          if response["error"] != 0:
37              # Cut the engine to make sure that we don't hit anything
38              egoVehicle.emergencyStop()
39          else:
40              # Update our various pieces
41              ...
```

Listing 4: The original application's networking code. It sends data asynchronously to the RSU and waits for only 10 milliseconds before continuing execution.

The first bug is that there is a highly optimistic estimation for waiting for a network response from the RSU. On Line 33, the CAV puts local fusion data into an asynchronous networking process running `processCommunications`, which sends data to the RSU through HTTP. However, the following sleep only waits for 10 milliseconds, as seen on Line 33. It is highly optimistic that the RSU execution of `global_fusion` between two CAVs and a round-trip response will finish within 10 milliseconds of sending the data. What occurs with this asynchronous send is a pipelining effect. The data it receives is actually the global fusion data from the previous iteration's run. A visualization of this can be seen in Figure 3.7.

To confirm this, we reran the application by serializing the network execution. This ensures that the CAV waits for the RSU to return from the current iteration's local fusion data before continuing. We allowed the CAV to run as fast as possible with this change. This change caused the original application's periodic iteration cycle time to slow down to 211.48 ms. This reaches parity with the speed found in

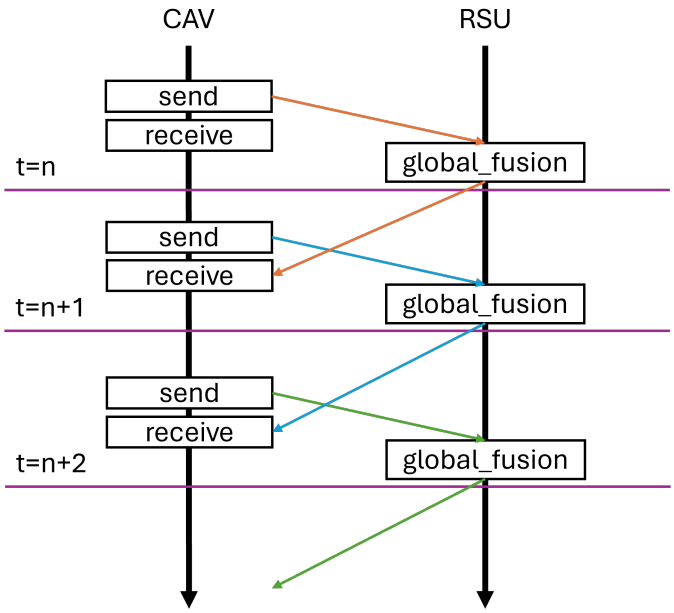


Figure 3.7: A time graph of execution between the CAV and RSU. At time $t = n + 1$, the CAV uses `global_fusion` data from time t. Data generated in the same iteration shares the same colors.

TTPython's initial testing. We set a periodic execution of 250 ms to avoid TTEH from executing at a higher rate due to tight deadlines. We expect a similar expectation to be set for the original application if it were updated.

The second bug we identified was a correctness issue in the asynchronous networking code. The process responsible for networking responses either sets a shared memory location as failed or successful, as seen on Lines 11 and 13 respectively. However, as the main process (on Line 36) only reads if the response has an error and does not invalidate if it uses it. This means that in an asynchronous execution, the next iteration can incorrectly use the previous reading's global fusion data. We saw in the Introduction (Section 1.1) that the `checkin` function takes up to 1 second before returning. As the application runs at a 125ms period, using stale data for 1 second before the HTTP response error updates can be dangerous.

3.2 Urban Flooding Network

The urban flooding (UF) network focuses on wide-area sensing for detecting flooding of intercity sewer systems [1]. Elizabeth Carter from Syracuse University is working closely with the United States Geological Survey (USGS) and the city of Syracuse to develop this application. Uneven water levels in a city sewage system can be caused by numerous external factors, such as impeding foliage or concentrated rainfall. Identifying sewage blockage is imperative as treatment prevents the backflow of sewage from violently expelling from house plumbing. The application combines thermal and optical imaging with location sensors (GPS and IMU) to identify where flooding occurs within the sensor network deployed across the city. If flooding is detected, the app will notify personnel to address the flooding. Furthermore, the device that detected the flooding will notify surrounding devices to increase their sampling rate to better gauge the water levels as the situation unfolds. The city of Syracuse plans to integrate this application

into its city infrastructure, which can provide a more stable environment in terms of power and internet capabilities, while the USGS is interested in more remote locations where cell reception can be difficult to find. As a member of the development team, our contribution to the application development includes creating system architecture diagrams for future deployments and adding new features in TTPython to accommodate for cost-effective hardware in sensor networks.

The urban flooding application consists of sensor boxes [53], LoRa [54] routers/gateways, and remote servers. A sensor box contains cameras, GPS, and an IMU. The autonomous version of the application is described as follows. The optical camera generates an optical and near-infrared image that is tagged with geolocational metadata from the GPS and IMU sensors. These images are combined with a long wave infrared image from the thermal camera, forming a multiband image. The multiband image is fed into a machine learning model to identify whether flooding occurs and is uploaded through LoRa (a radio communication technique) to a remote server. The LoRa gateway acts as an intermediary between the sensor box and server.

The UF case study is motivated by the following research question.

RQ **What are the design patterns that developers use when using TTPython's distribution and timing abstractions?**

The application is an ideal real-world testbed for TTPython as it provides unexplored system requirements for TTPython. Specifically, the UF app motivated the development of `TTSingleRunTimeout` as seen in Section 2.2.3, support for multitenancy, and encoding unique modality execution based off of environmental factors. The two latter aspects are discussed in a later section after discussing the proposed system architecture of UF app and the code of the TTPython application.

3.2.1 System Architecture

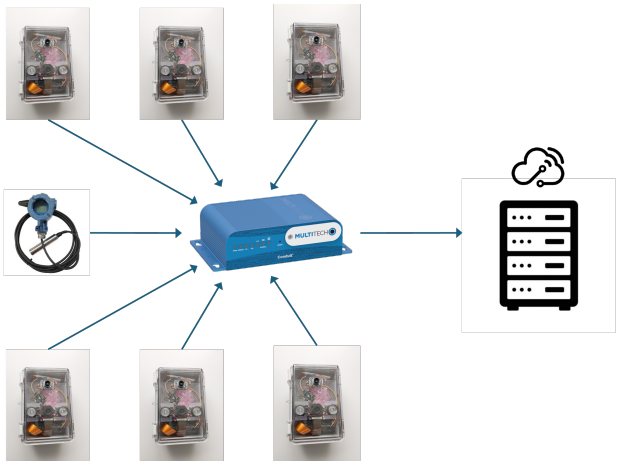


Figure 3.8: A neighborhood in the UF app. 6 sensor boxes and a transducer talk to a LoRa gateway that forwards data to a server.

The initial goals of the project have expanded from neighborhood analysis to multi-city deployments. The project development is planned to go through three stages: device-level implementation, neighborhood deployment, and city-wide deployment. As the current state of the application's development is

still at the device level, the code artifact analysis will be limited to device-level interactions and development. A discussion on the system design of a neighborhood deployment is presented below. We argue that these graphical representations are amenable to TTPython's graph execution model and highlight the ease of taking high-level requirements and representing them in TTPython. We first present the system architecture of a neighborhood in a city-wide deployment.

We define a *neighborhood* to be six sensor boxes, a submersible water level transducer sensor, a LoRa gateway, and a remote server. These sensor boxes and transducers use LoRaWAN to send data wirelessly while minimizing energy cost. The LoRa gateway sends LoRa radio packets to a remote server that interprets LoRa packets into more traditional formats, such as IPv6. The remote server also hosts TTPython code that coordinates communication between sensor boxes. We have opted for a centralized routing table to allow for flexible neighborhood deployments, as routing table changes can be done much more easily on the server side rather than on in-field devices. The six sensor boxes and transducers are unique per neighborhood, while the LoRa gateways and remote servers may be shared across neighborhoods. The transducer passively produces data and offers an API to change its sensing rate. The transducer is not programmable and, as such, does not support TTPython. Its interactions are out of scope for the thesis and can otherwise be modeled as external input or output per graph execution.

A sensor box running the autonomous version of the application has multiple modes. The state transition diagram is shown in Figure 3.9.

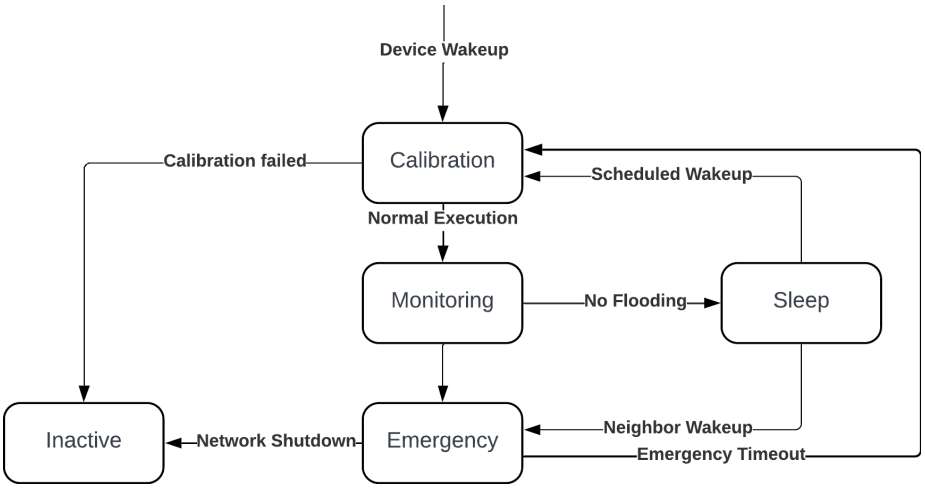


Figure 3.9: A sensor box's state transition graph.

When woken up, a sensor box starts with a calibration step. These steps are covered in Figure A.1. If calibration is successful, it will move to a monitoring phase that samples the surrounding locale to check if there is flooding. If no flooding occurs, the sensor box will go to sleep to conserve energy. If flooding is detected, the device moves to emergency mode, alerts nearby sensors to go to emergency mode, and constantly samples the environment while it is flooding. It keeps track of the area of the flooded location in the image. Once flooding stops, the sensor box will send this tracked area to the server, which displays the data on a dashboard.

The monitoring phase is shown in Figure 3.10. The graph is a visual representation of the autonomous version of the application seen later in Section 3.2.2.

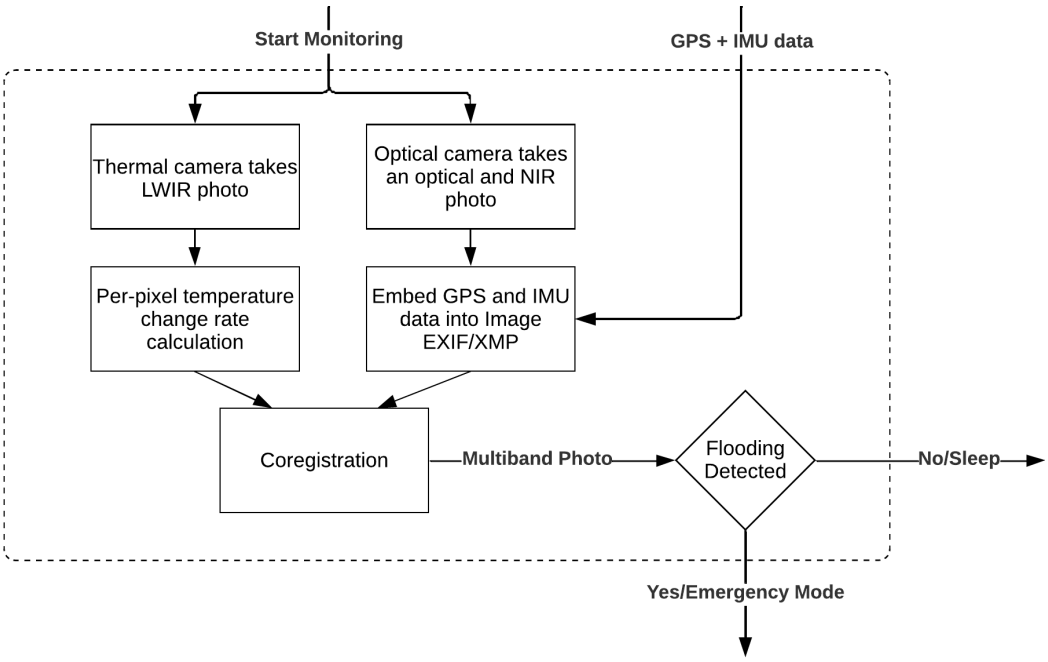


Figure 3.10: Monitoring mode in the state transition diagram.

3.2.2 Code Artifact Analysis

The urban flooding application currently has two versions available. The first one is an on-demand version where a user takes a camera box and presses a pushbutton to take a picture. This application version is being used to accrue optical and thermal images for training data. This version is a subset of the autonomous version described above, as it does not run code pertaining to flooding classification in an image. The autonomous version is described in Listing 5. We discuss the main TTPython abstractions found in the code below. These abstractions answer RQ.

This example already highlights many timing requirements already discussed in the prior Section 2, including the composition of the `TTFinishByOtherwise` and `TTSingleRunTimeout` construct on Line 16. We instead focus on new additions to TTPython brought by the UF application: design patterns with STREAMified nodes for exceptional time handling, varied execution modes with the `TTPersistent` flag, and alternative networking handling.

`@STREAMify` Control Flow Design Pattern

TTPython's exceptional time handler `TTFinishByOtherwise` depends on its control tokens to specify when the deadline occurs. This means that each instance of `TTFinishByOtherwise` (Line 16) requires SQs (such as on Line 14) to generate these deadline control tokens. The question is then when `READ_TTCLOCK` should run. The intent is difficult to capture in the dataflow graph setting. In an imperative language, deadlines would be generated at the start of the periodic control loop. However, TTPython encodes periodicity at the firing rule level. `READ_TTCLOCK` could use the output of a STREAMified function (such as `get_time` on Line 5), but the deadline would begin from when the output token of the STREAMified function was produced. This would exclude the STREAMified function in the calculation of the deadline, which might not be the programmer's intent. A comparable translation of generating deadlines at the start of the periodic loop is seen in Listing 6. The difference here is that the STREAMi-

```
1 @GRAPHify
2 def ttmain(trigger):
3     with TTClock.root() as root_clock:
4         # alternative syntax for generating deadlines
5         loop_start, dirname = get_time(trigger,
6                                        TTClock=root_clock,
7                                        TTPeriod=10_000_000,
8                                        TTPhase=0,
9                                        TTDataIntervalWidth=1_000_000)
10        photo = take_optical_images(trigger, dirname, TTPersistent=True)
11
12        lepton_file_name = get_thermal_image(dirname)  # flaky call
13
14        deadline = READ_TTCLOCK(loop_start, TTClock=root_clock) + 5_000_000
15
16        lepton = TTFinishByOtherwise( # composition of timing constructs
17                        lepton_file_name,
18                        TTTimeDeadline=deadline,
19                        TTSingleRunTimeout(lepton_planb(),
20                                           TTTimeout=10_000_000),
21                        TTWillContinue=False)
22
23        # ensures that multiple calls are sequentially handled
24        coreg_state = coregistration(dirname, lepton, photo,
25                                     TTPersistent=True)
26        seg_result = segformer(dirname, coreg_state, TTPersistent=True)
27        bitmap = compress_bitmap(seg_result) # bitmap sent over LoRa radio
28        lora_return = lora_token(bitmap) # bitmap received by server
29        y = call_shutdown(lora_return)
```

Listing 5: The GRAPHified main function of the autonomously deployed version of the UF app.

fied function has been moved to an `identity` function from the `get_time` function. The `identity` function's purpose is to move the control loop to an explicit SQ. As a simple function, `identity` would take negligible time to execute. Its output token can then be used to start the original loop and generate deadline tokens at the same time. Referring back to Listing 5 on Line 5, we can then see the shorthand notation for describing the timing nuance. `get_time` returns `dirname` after its execution to generate a pathname dependent on the current time. This is used to store the optical and thermal images. Right before running `get_time`, TTPython will emit a token (`loop_start`) so that `deadline` can be computed from the start of when `get_time` runs. The line is translated to the Lines 2-7 seen in Listing 6.

The compiler needs two new rules to account for this change. First, we modify the DV firing rule in the STREAMified SQ (v_f) to emit a control token through c' before executing its internal function f. We denote this addition as DV$'$. c' has its own `resample` node (v_c) to update the emitted token's timestamp to the current iteration. We label the updates to the STREAMify rule below in red.

```
with TTClock.root() as root_clock:
    loop_start = identity(trigger,  # identity func to signal period start
                          TTClock=root_clock,
                          TTPeriod=10_000_000,
                          TTPhase=0,
                          TTDataIntervalWidth=1_000_000)
    dirname = get_time(loop_start)  # now SQified as periodic info moved

    deadline = READ_TTCLOCK(loop_start, TTClock=root_clock) + 5_000_000
```

Listing 6: An equivalent execution of the alternative `@STREAMify` control token syntax in Listing 5 on Line 5.

$$\frac{\begin{array}{c}\forall i = 1..n.\ e_i \mapsto \mathsf{V}_i; x_i \\ x_i' \text{ fresh} \quad x_i^g \text{ fresh} \quad x_i'' \text{ fresh} \\ v_i = \langle [x_i'] = \texttt{repeat}(x_i, x_i^g), \texttt{Immediate}, \langle -\infty, [-\infty, -\infty] \rangle \rangle \\ v_i' = \langle [x_i''] = \texttt{delay}(x_i', \texttt{first\_offset} = n_2, \texttt{offset} = n_1), \texttt{DV}', \cdot \rangle \\ v_f = \langle [c', x'] = f(x_1'', \cdots, x_n''), \texttt{DV}', \cdot \rangle \quad c' \text{ fresh} \quad x' \text{ fresh} \\ v_i^r = \langle [x_i^g] = \texttt{resample}(x_i'', \texttt{start\_offset} = -n_3/2, \\ \qquad \texttt{end\_offset} = n_3/2), \texttt{DV}, \cdot \rangle \\ v_r = \langle [x] = \texttt{resample}(x', \texttt{start\_offset} = -n_3/2, \\ \qquad \texttt{end\_offset} = n_3/2), \texttt{DV}, \cdot \rangle \quad x \text{ fresh} \\ v_c = \{ [c] = \texttt{resample}(c', \texttt{start\_offset} = -n_3/2, \\ \qquad \texttt{end\_offset} = n_3/2), \texttt{DV}, \cdot \} \quad c \text{ fresh} \end{array}}{\begin{array}{c} f(e_1, \cdots, e_n, \texttt{TTPeriod=}n_1, \\ \texttt{TTPhase=}n_2, \texttt{TTDataIntervalWidth=}n_3) \mapsto \\ (\bigcup_{\forall i=1..n.} \mathsf{V}_i \cup \{v_i, v_i', v_i^r\}) \cup \{v_f, v_r, v_c\}; (c, x) \end{array}}\ \texttt{STREAMify}' \textit{ Func} \tag{3.1}$$

The Assign statement rule also needs to change with this new syntax. This rule will only be applied if the top level expression is a `STREAMify`$'$, as it is the only expression that returns multiple names. We highlight the changes in red once again.

$$\frac{e \mapsto \mathsf{V}; (c', x) \quad \theta = \{(x \to y)\} \quad \theta' = \{(c' \to c)\}}{c, y = e \mapsto \mathsf{V} \circ \theta \circ \theta'; y}\ \textit{Assign'}$$

Varied SQ Execution Modes

The `TTPersistent` flag seen on Line 25 in Listing 5 exposes SQ runtime considerations to the programmer. A SQ is composed of a Synchronization, Computation, and Communication phase as described in Section 2.5. SQs without side effects can be spawned concurrently without producing issues. TTPython's execution strategy by default assumes that SQs do not have side effects. This increases throughput and adheres to tagged-token dataflow graph semantics. However, not all SQs are written in a stateless manner. Some SQs access OS resources and continue to hold those resources over their lifetime. Other SQs want

to be executed in a sequential manner. For example, `coregistration` uses many image processing techniques in the OpenCV library [9] that are computationally expensive. This SQ takes a long time (40 seconds) in comparison to its periodicity (10 seconds). This means that the SQ will be called again in the next iteration before its prior instance has finished. An eager approach to spawning a SQ execution for each period can strain the device, especially if run on cost-effective hardware. The `TTPersistent` flag specifies to the TTPython runtime to only allow one instance of the SQ to run at a time and synchronizes all periodic calls to that instance.

This construct may seem similar to `TTSingleRunTimeout` in the sense of only allowing one instance to occur, but its use cases are different. `TTSingleRunTimeout` operates over a subgraph while `TTPersistent` is assigned per SQ. `TTSingleRunTimeout` also discards any calls to its expression if the expression is already being run, while `TTPersistent` SQs will queue up these calls. `TTPersistent` allows the programmer to coordinate hardware and OS resources *within* an application, but the UF application suggests that these need to be shared across applications. Further discussion of this sharing is found in Section 3.2.3.

LoRa Handling

One advantage TTPython offers is network abstraction through the dataflow graph. As the code in Listing 5 only describes the monitoring mode of the device, it does not capture the networking challenges TTPython handles in the UF application at the neighborhood level. At the neighborhood level implementation, the code on Line 28 would be on the sensor box while Line 27 would be on the server side (Figure 3.8). Without TTPython, resolving the receiver of a LoRa packet would be much more difficult. To minimize the payload size, the LoRaWAN protocol does not provide a standardized way to identify the sending or receiving devices. This is quite different from an IP-based protocol, where that data is already encoded in a packet. This means that programmers need to distinguish their LoRa packets into predetermined message types ad hoc per device. The burden lies on the programmer to encode the necessary information to specify what type of data is in the payload, decode the packet when it arrives at the server, and determine where to forward the data once it has determined what it is.

The advantage TTPython gives in this scenario is how it encodes data over the network. Data is sent in the form of `TTTokens` with a `TTTag` to specify which SQ, device, and port it should be sent to. By homogenizing data types with `TTToken` and abstracting its sending location as a symbolic reference (uniquely defined by `TTTag`), the encoding, decoding, and forwarding procedure required by LoRa vanishes. Given a `TTToken`, TTPython first resolves symbolic device names with IP addresses and forwards them to the correct device. Once the `TTToken` reaches the correct device, TTPython looks up the SQ and places the `TTToken` into the SQ's port number.

3.2.3 Future Case Study Directions

As the development at Syracuse University is still underway, there are open questions on TTPython unresolved by this thesis. We discuss these below.

Multitenancy

TTPython as a middleware can support different applications running on the same hardware. Although applications can be designed in a monolithic manner, doing so makes it difficult for developers to design in a modular fashion. For example, one of USGS's requirements for the project includes the feature to take images on demand for deployed sensor boxes. If flooding has been detected by a sensor, a human should

have the option to request an image from it for manual inspection. Deployed sensors can often be inaccurate due to environmental circumstances, such as the camera having a dirty lens or being displaced from its expected location. Having a human manually check if flooding occurs from on-demand images mitigates false positives in flooding classification and prevents unnecessary human deployment. This requirement would be implemented as a separately designed application running alongside the current autonomous version of the application. The interest lies in how these applications share hardware requirements and how the OS provides these accesses. For example, the Raspberry Pi's camera API provides a unique handle to the first process that it interfaces with. This means that if other user processes attempt to query the camera while the original process still holds it, it will fail. These applications unknowingly interact with each other as they require an optical image from the Raspberry Pi camera. The key insight is that in a real-time setting, time can uniquely define data. If these two applications asked for an image at the same time, a single image could satisfy both applications. Both these applications have a SQ related to interfacing with the optical camera. During compile time, the programmer can specify to TTPython that a particular SQ should be treated statically with respect to other applications. During runtime, the separate applications call the same SQ to handle accessing the camera, which allows multiple applications to share the camera.

These ideas of having unique access to hardware appear in other aspects of TTPython as well. `TTSingleRunTimeout` introduces synchronization at the subgraph level within a graph, while multitenancy takes this idea and applies it across multiple graphs. `TTSingleRunTimeout`'s use case was to ensure that only one attempt of resetting the thermal camera was allowed during a period of time, while multitenancy attempts to synchronize calls to the optical camera when used across the autonomous and the on-demand versions of the application.

3.2.4 Modality Execution

As discussed in Section 3.2.1, a device's execution can switch between monitoring and emergency mode. The differences between each mode are small. In both modes, the device still takes images and runs them through the flood detection model. The frequency at which it runs differs, though. The device should continually run when the area is flooding. Furthermore, in emergency mode, the device needs to send an image of the maximum flooded area when the device deems that it is no longer flooding. In general, the differences are found in the periodicity and the data it sends to the server. In TTPython's current implementation, these states would need to be encoded as separate graphs. Periodicity is statically encoded in the SQs. However, we could take advantage of the code that these states share. The graphs would share a majority of the SQs they run, albeit with slight differences in periodicity and data handling. Future work would look into encoding how to encode different modalities of an application and to take advantage of the overlap of SQ and graph behavior.

3.3 Case Studies Conclusion

In these two case studies, we saw TTPython used to implement two realistic DT applications. TTPython's macroprogramming approach made networking components easy for users of TTPython to handle. Its abstractions in the SI application reduced 340 LoC to 5 LoC in TTPython. Swapping between different network interfaces is also easy for the user, as seen in the UF application. The custom encoding and decoding requirements required for LoRa packet management are generalized for the user for the normal case with TTPython's `TTToken` encapsulation.

We also identified three bugs associated with concurrency in SI application. These appeared in concurrent code associated with asynchrony. The original application did not include rigorous checks to ensure the data being used was synchronized. We saw bugs in the networking code leading to an unnecessarily long timeout period, unintended pipelining effect, and staleness of data. When the CAV tries to send information of the network through a HTTP request, it waits up to 1 second before timing out. This is long in comparison to its 8 Hz periodic loop and was unintended by the original designer. TTPython's dataflow graph execution semantics are inherently parallel and remove overhead from the programmer to interface with low-level concurrency libraries. We believe that in scenarios where concurrency is used not for performance but for availability, concurrent code should be designed in a functional/dataflow graph-like manner. This approach separates infrastructure code from application code and focuses on correctness. Debugging distributed applications in a real-time setting can be challenging, so removing overhead for the programmer by reducing infrastructure code can be beneficial.

Chapter 4

Qualitative User Study

We conducted a user study to observe what programmers unfamiliar with DT applications find difficult when writing them. We wanted to observe how DT applications would be created from scratch in TTPython and vanilla Python. Vanilla Python requires much more scaffolding and code, so we provided custom libraries and a message broker to abstract networking. Participants created two applications derived from the case studies: the smart intersection and urban flooding. These two applications are written in both TTPython and vanilla Python with library support for timing and distribution, namely the Python `time` library and a message broker, RabbitMQ. We chose RabbitMQ for the user study, as it is a popular, open-source message broker and requires less setup compared to creating HTTP web servers, which were used in the case studies. We ran this study with four programmers.

4.1 Study Design

The user completes five tasks in total for each application. These tasks focus on **asynchronous data generation**, **data synchronization**, **networking**, **time-triggered exception handling**, and **code evolution**. We first present the background of each application before discussing the motivation for these task selections.

4.1.1 Smart Intersection

The smart intersection architecture does not differ much from its original implementation. The user works with two CAVs and a single RSU. The major difference is that the application the user first implements has the CAV's route planning on the RSU. This motivates the code evolution task by requiring the user to move it back onto the cars. The movement of code requires users to also rewire the exceptional time handlers on the CAVs. We discuss this in more detail in Section 4.6.

4.1.2 Urban Flooding Network

In the urban flooding network, we decided to split the sensor box into an optical camera device, a thermal camera device, and a router. This was both in part to reduce the amount of data streams necessary for the user interface within the data synchronization task and to produce a more interesting system architecture for the networking task. The optical camera device is assigned functions related to generating an optical and near-infrared thermal image and tagging them with geolocational data. The thermal camera device

takes a long-wave infrared image and combines these images. The router hosts the ML model and uploads it to the remote server.

4.1.3 Task Choice

We selected tasks representative of realistic settings when designing the SI and UF application. We chose the four characteristics of distributed, time-sensitive applications as our first four tasks and added one task examining what changes would be necessary when modifying system architecture in a completed application. First, DT applications rely on **asynchronous data generation** for interesting, autonomous devices. Data generation and handling are asynchronous to provide code that is robust to unresponsive hardware or networks. The smart intersection features CAVs that periodically generate data from camera and LIDAR sensors. The urban flooding network uses two cameras, a GPS unit, and an IMU unit to detect flooding in the environment. Second, **data** in DT applications need to be **synchronized** before use. Sensor data should be timestamped when they are generated, and operations on data need to account for this timestamp. In practice, data is not generated simultaneously, so data must be synchronized to ensure those operations are meaningful. Third, these applications require multiple devices to coordinate over the **network**. In the smart intersection, the CAVs talk with the RSU and vice versa for local and global sensor fusion. The urban flooding network combines the images generated from the thermal and optical cameras into a machine learning model hosted on different devices. Fourthly, devices need to respond if the network or sensors become unresponsive with **time-triggered exception handling** code. Within a specified deadline, if a CAV hasn't heard back from the RSU, it should take preventative measures, such as applying the brakes. If one of the cameras in the urban flooding network becomes unresponsive, the device can attempt to power cycle the camera unit to reset the connection and hardware state. Lastly, these applications can change over time due to changing specifications and unforeseen complications found during deployment. For example, in the smart intersection, redesigning the flow of information in the system architecture can increase CAV autonomy. Specifically, the CAVs in the smart intersection only need to stop if they are at the intersection and could autonomously move if there are no other CAVs in the area. In the urban flooding network, device compute capabilities can widely vary due to the magnitude of deployment. Running a machine learning model on cost-effective devices such as a Raspberry Pi can be energy- and time-intensive. Instead, it might be cheaper to send the image over the network to a server with stronger computing power to handle the classification task. **Code evolution** is a natural part of DT application development.

4.1.4 Recruitment

We require participants to have at least 2 years of experience in programming, 6 months of experience with Python code, and experience working with concurrency through a systems course or equivalent experience. The user study focuses on completing coding tasks, so we require participants to have prior programming experience. To implement a vanilla Python application similar to TTPython, the participant needs some systems background in understanding concurrency techniques, such as queues and processes. We ask participants to specify their prior programming experience in Python, overall programming experience, and experience with using message brokers. During the sign-up process, participants take a short quiz on their understanding of timing and process interactions to determine whether participants would struggle with the timing and concurrency ideas in the study. Participants also specified if they had prior experience with using RabbitMQ.

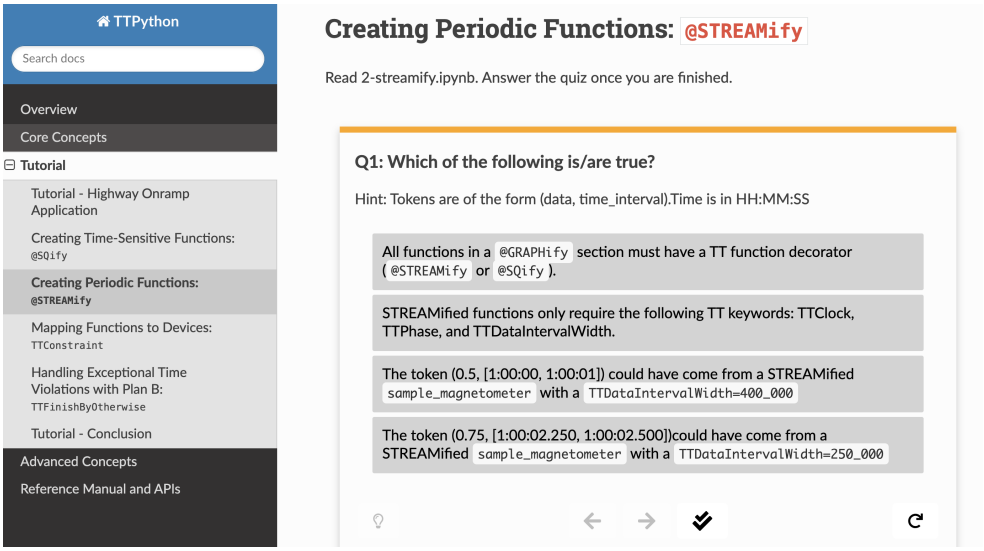


Figure 4.1: Sample quiz on the `@STREAMify` section in the TTPython tutorial.

4.1.5 Tutorials

For each system, we provided a tutorial explaining concepts used later in the tasks. The tutorials were contained in a Jupyter notebook to simplify environment setup and code execution. Participants completed quizzes at the end of each section to improve retention and learning of the material. These quizzes were either included in the Jupyter notebook, with answers verbally confirmed with the study administrator, or taken with multiple-choice questions from a web-based form. Participants were allowed to ask questions about the material.

TTPython Tutorial

The TTPython tutorial, as seen in Figure 4.1, is divided into four main sections that correspond to the first four tasks. The tutorial motivates TTPython with participants creating a highway on-ramp application. Participants are first introduced to TTPython's graph execution semantics and data synchronization primitives with `@SQify`. They then create periodic streams with `@STREAMify`. The tutorial then introduces the `TTConstraint` construct to handle SQ assignments to devices, and thereby abstracts networking from the programmer. In the final section, participants learn about exceptional time handling with the `TTFinishByOtherwise` construct. Participants were limited to 45 minutes to complete the tutorial. The length is to give participants enough time to learn the dataflow graph semantics of TTPython.

Python Tutorial

The tutorial for the vanilla Python implementation, as seen in Figure 4.2, is composed of three sections: timing, concurrency, and RabbitMQ. This tutorial provides a background for users to implement the data structures and logic required for the tasks. As we are measuring the difference between two systems achieving the same output per task, the vanilla Python tools have less material to cover. The timing section introduces the primitives of Python's time library and provides a framework for creating periodic code. The concurrency section explains how to create processes in Python and the Queue API for interprocess communication. This section also describes how to make non-blocking reads to the queue, which the

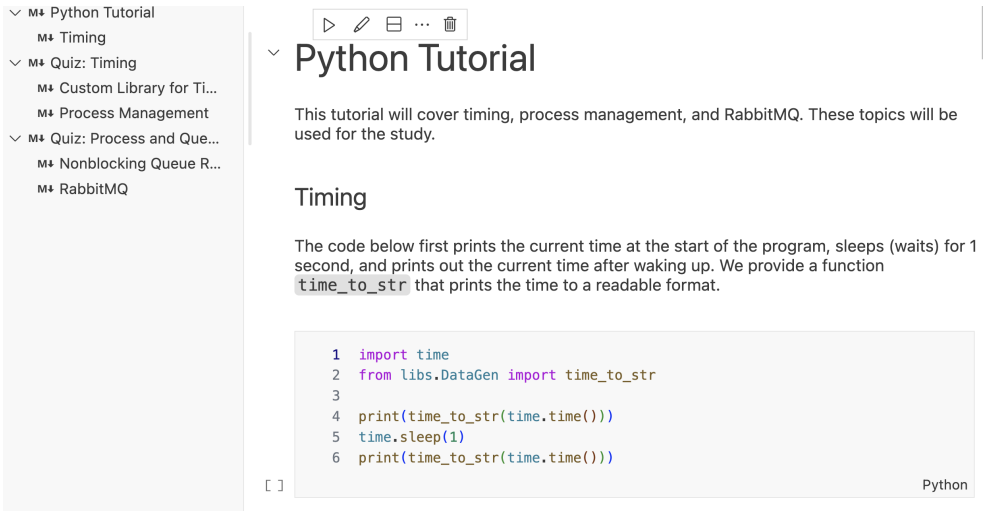


Figure 4.2: Sample Jupyter notebook tutorial introducing Python time and concurrency concepts.

participant uses to access multiple, asynchronous data streams in the first task. The final section teaches the sending and receiving sample code found in the official 'Hello World' RabbitMQ documentation. Participants were limited to 20 minutes to complete the tutorial. The length is shorter in comparison to TTPython, as there are fewer dependencies required to complete the Python tasks. The tutorial is meant to introduce Python's API on the timing and concurrency standard library to give participants a fair level of understanding before attempting the coding tasks. The vanilla Python tasks focus on programmers building infrastructure with these lower-level libraries to achieve the same tasks in TTPython.

4.1.6 Study Protocol

Participants write an implementation for both the SI and UF application: one with TTPython and the other with vanilla Python. Each application is broken down into five tasks: **Asynchronous Data Generation**, **Data Synchronization**, **Networking**, **Time-Triggered Exception Handling**, and **Code Evolution**. Each task is presented in a Jupyter Notebook to combine the task description and the code environment seamlessly. Participants either write their code directly in the Notebook or in files hyperlinked in the Notebook to complete their task. As participants use both TTPython and vanilla Python in the study, we control the type of application and the order of using each tool. Participants are randomly assigned an ordering and a tool selection for each application. For example, Participant 1 writes the SI application first in TTPython and the UF application in vanilla Python, while Participant 2 writes the UF application first in TTPython and the SI application in vanilla Python. This gives us four different possible static protocols that a user could experience (either TTPython or vanilla Python first, and which application is written first). Our four participants were counterbalanced so that each participant ran a different permutation of the tool order and implementation.

For each code task, the participant is briefed with an explanation of the task's purpose. Each code task has the same explanation provided, except for where the terminology would differ. For example, for concurrency, the vanilla Python implementation requires the `multiprocessing` library while TTPython uses the SQ abstraction.

When completing the tasks, the user is directed by TODOs and specific directions on what to implement. These TODOs can appear mechanical as they are precise and do not leave much for creative

interpretation. This was done to reduce the difficulty of the tasks, as participants are expected to implement most of a the SI or UF app within their 1 hour and 30 minutes of allotted time. Many code implementations are also dependent on the system architecture of the application. For example, data synchronization is only necessary when data is used between different asynchronous sources. This is unnecessary when data has already been synchronized. Network topology and process management make it difficult to identify which sources are asynchronous, so we explicitly mention when it is required in the TODOs for vanilla Python.

Participants have access to a code cell in the Jupyter Notebook to run their programs and a sample solution output to compare their program output to. Each code task has a hard cutoff of 25 minutes to prevent participants from spending too long on any task. The participant is given a warning at 20 minutes to finish up before moving on to the next task.

4.1.7 Post Study Questionnaire and Interview

After completing both applications, participants filled out a questionnaire on their experiences with each system. These questionnaires used a combination of System Usability Scale (SUS) [11] questions on a 5-point scale and the NASA Task Load Index (TLX) [28] on a 10-point scale. SUS is an industry-standard self-assessment to quantify the usability of a system. NASA TLX is widely used to measure demands, performance, and effort when performing a task. We also had a short semi-structured interview after the questionnaire to learn about their experience with using both systems. These questions focused on the challenges that arise from the distributive, time-sensitive nature of the applications.

- How did your experience of using the tools compare from one another?
- What did you feel was difficult with specifying time/time-triggered exception handling/distributed development in these two applications?
- What are the challenges when using TTPython/vanilla Python with RabbitMQ?
- Which tool did you prefer overall?

4.1.8 Research Questions

Our study answers the research questions listed below.

RQ 1 **What are the challenges developers experience in vanilla Python/TTPython when writing**...

a) **timing code with concurrency?**

b) **networking with concurrent code?**

c) **time-triggered exception handling code?**

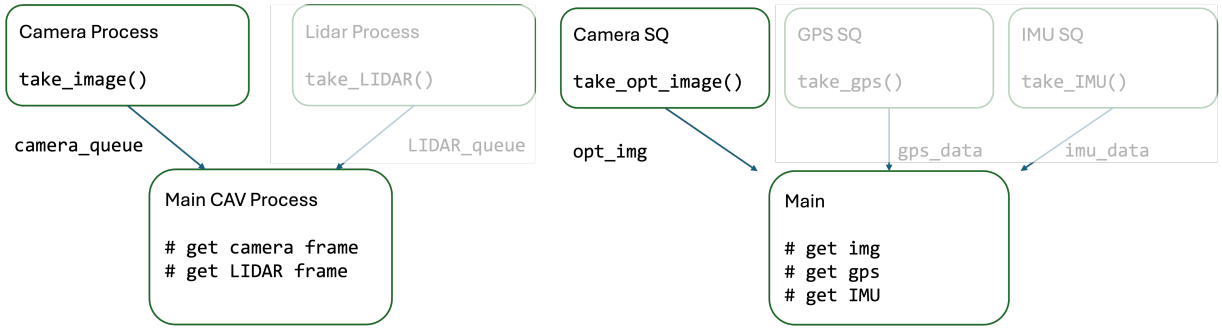
RQ 2 **How do programmers unfamiliar with DT applications react when using TTPython's timing and distribution abstractions?**

4.1.9 Data Analysis Methodology

We collected data by capturing the screen and audio throughout the user study. We did not require participants to engage in think-aloud; however, we did ask participants about their thoughts when they seemed visibly confused. For each recording, we transcribed each participant's interview first with Whisper AI [52] that was later manually cross-checked with the audio. We took this data to perform thematic analysis [10] and qualitative coding. Our task selection provided the framework for the questions we asked during the semi-structured interview and the codes used to select our data and quotes. Each participant's performance on tasks was compared to the sample solution provided. Testing the vanilla Python ADG and DS tasks was more granular due to the modularity of the code, so participants could create partial solutions. The NTWK, TTEH, and CE tasks have nontrivial implementations, so success depended on whether a participant's solution matched the sample solution's output. Since the tasks were designed on TTPython's base constructs, their task success was also based on matching outputs for the sample solution. We discuss the success of each task in its respective subsections.

4.2 Task: Asynchronous Data Generation (ADG)

The first task has participants create asynchronous, periodically executing code. The sample system architecture is shown below. The system architecture explains the scope of the ADG Task while explaining the other data streams that will be used later in the study.



(a) Vanilla Python SI App (b) TTPython UF App

Figure 4.3: The system architecture presented to introduce the ADG Task.

Participants are tasked to first create a minimal DT application. This involves creating a main function with one asynchronous data stream. We chose to use a camera as the main data stream to implement, as it is used in both the SI and UF applications. Specifically,[1] the participant had to create a periodic control loop call to a camera device and capture timing information with it. The code to get an image from the camera was provided as a library call as starter code.

4.2.1 TTPython Implementation

In TTPython, creating asynchronous data-generating code requires two steps. First, the programmer has to decorate a function with the `@STREAMify` decorator to make it valid for use with the TTPython compiler. Secondly, the programmer uses the function in the compiled graph function (decorated by `@GRAPHify`)

[1]Refer to A.1.3 to see the instructions given and the starter code.

and calls it with special TT keyword arguments to specify its periodic timing requirements. An example solution is shown below in Listing 7.

```
1 @STREAMify  # solution
2 def take_opt_image(trigger):
3     ...
4
5 @GRAPHify
6 def main(trigger):
7     with TTClock.root() as root_clock:
8         start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 1_000_000
9         sampling_time = VALUES_TO_TTTIME(
10             start_time, GET_INFINITY(trigger, TTClock=root_clock))
11         sample_window = COPY_TTTIME(trigger, sampling_time)
12
13         ###### T Task 1:
14         # TODO: Make `take_opt_image` periodic and call it with a period of 2
15         # seconds. Have `take_opt_image` use `sample_window` as its trigger.
16         # Use a TTDataIntervalWidth of 500_000 microseconds and set TTPhase=0.
17         images = take_opt_image(sample_window,  # solution
18                                 TTClock=root_clock,
19                                 TTPeriod=2_000_000,
20                                 TTPhase=0,
21                                 TTDataIntervalWidth=500_000)
```

Listing 7: The TTPython solution for the ADG Task in the UF app. The highlighted sections show a sample solution for the task.

In the code above, users have to add the `@STREAMify` decorator on Line 1 in Listing 7. They also call the STREAMified function `take_opt_image` in the `@GRAPHify` section with `TTClock`, `TTPeriod`, `TTPhase`, and `TTDataIntervalWidth`. We provide participants with the period, phase, and `TTDataIntervalWidth` as they are application-specific arguments and are dependent on the environment in which the application is deployed. These parameters are also given for the vanilla Python version.

4.2.2 Python Implementation

The vanilla Python is more involved as participants must work with Python's concurrency and timing primitives to recreate what TTPython provides. To mimic SQ execution, vanilla Python has to run the corresponding function in a different process. Participants create a child process responsible for producing images from the camera device with timing information. These are then sent to the main process coordinating between all the devices' data generators. We provide a solution in Listing 8.

```
1 def approximate_data_timestamp(func, tolerance):
2     before = time.time()
3     data = func()
4     after = time.time()
5     taken_at = (before + after) / 2
```

```
    time_interval = [taken_at - tolerance, taken_at + tolerance]
    return (data, time_interval)

def take_opt_image(opt_cam_queue: mp.Queue, start_time, period,
                           tolerance):
    wait_until_start = start_time - time.time()
    if 0 < wait_until_start:
        time.sleep(wait_until_start)

    target = start_time

    # Init the camera class
    opt_cam = DataSources.Camera()

    while 1:
        if target <= time.time():

            # Take the camera frame and process
            image, timestamp_interval = approximate_data_timestamp(
                opt_cam.take_camera_frame, tolerance)

            print('sending: '
                  f'{(image, [time_to_str(t) for t in timestamp_interval])}')

            opt_cam_queue.put((image, timestamp_interval))

            print(f'received: {opt_cam_queue.get()[0]}')

            # New target
            target = target + period
        time.sleep(.001)

def opt_main():
    start_time = time.time() + 1
    period = 2
    tolerance = 0.5
    fallthrough_delay = 1

    ### P Task 1-2: Spawn the camera processing process.
    opt_cam_queue = mp.Queue()
    opt_cam_process = mp.Process(target=take_opt_image,
                                 args=(opt_cam_queue, start_time, period,
                                       tolerance))
    opt_cam_process.start()
    ### P Task 1-2

    ### P Task 1-2: Wait until the start_time to start executing
    wait_until_start = start_time - time.time()
    if 0 < wait_until_start:
        time.sleep(wait_until_start)

    next_time = start_time
```

```
58      ### P Task 1-2
59
60      ### P Task 1-2: make a loop periodic
61      while True:
62          curr_time = time.time()
63          if next_time <= time.time():
64              fallthrough = curr_time + fallthrough_delay
65
66              photo_came = False
67              while time.time() < fallthrough and not photo_came:
68                  try:
69                      while (time.time() < fallthrough
70                             and not opt_cam_queue.empty()):
71                          photo, interval = opt_cam_queue.get_nowait()
72                          photo_came = True
73                  except queue.Empty:
74                      pass
75
76              print(f'received: {photo}')
77
78              last_next_time = next_time
79              while next_time <= time.time():
80                  next_time = last_next_time + period
81                  last_next_time = next_time
82
83          time.sleep(.001)
84      ### P Task 1-2
```

Listing 8: The vanilla Python solution for the ADG Task of the UF app. The highlighted sections show the additions for a sample solution.

The participant completes four subtasks. The first subtask timestamps the output for a given function as seen on Line 1. We require this timestamping procedure with function output to simulate working with 3rd-party libraries that do not inherently provide timestamped data. The participant then creates a control loop to periodically generate image samples. This implementation is open-ended for the programmer to design since periodicity can be implemented in different ways by Python's `time` library. We are particular in specifying that their periodic loop should continue to execute periodically based on the start of execution. This is to prevent the drifting of the periodic loop. For example, say we want to generate an image every 2 seconds. The code execution for taking an image and sending it over a queue takes a nontrivial amount of time. This should not affect the high-level specification of producing images every two seconds. The next task is to send the timestamped data through an ITP queue to the main program. The main task interfaces with all the different data streams, as seen in Figure 4.3. The final task is to create a periodic control loop and accept data from the child process when it becomes available. This involves understanding how to call the ITP queue in a nonblocking manner. The participant wraps this under a timer that tries to read from the queue until a deadline has passed.

4.2.3 Observations

The main challenge found in the TTPython section is for participants to use the `@STREAMify` annotation correctly with its TT keywords. In general, participants referred back to the tutorial to review the required

keyword arguments. Participant 3 (P3) was initially confused by using the `@SQify` decorator before switching to the `@STREAMify` decorator. We believe this confusion is caused by the out-of-order nature of the tutorial. The tutorial is structured in the following order for the TT constructs with `@SQify`, `@STREAMify`, `TTConstraint`, and `TTFinishByOtherwise`.

For vanilla Python, half of our participants had difficulties with ensuring that their periodic loops didn't drift over time. The standard definition for a periodic execution is to have its next execution start after a defined time interval from the beginning of the previous iteration of its control loop. For example, if the body of a loop takes 0.5 seconds to execute for a target period of 1 second, it should run its execution starting from time (t=0, t=1, t=2, ...), not (t=0, t=1.5, t=3, ...). In our vanilla Python tutorial, we did not provide participants with a periodic implementation. We instead introduced the concept of using an infinite **`while`** loop with sleep calls that almost approximates a periodic call. We wanted participants to figure out the logic change required to account for the body's execution time. We quizzed participants on this timing nuance in the vanilla Python tutorial, so participants were well aware of this definition. Even with this knowledge, two of our participants wrote an incorrect periodic loop. We examine a buggy code example written by a participant below in Listing 9.

```
1 period = 2
2 while True:
3     timestamped_data = approximate_data_timestamp(
4         opt_cam.take_camera_frame, tolerance)
5     time.sleep(period)
```

Listing 9: Buggy implementation of periodicity.

```
1 period = 2
2 target = start_time # calculated from the start of the program
3 while True:
4     if target <= time.time(): # polls repeatedly
5         timestamped_data = approximate_data_timestamp(
6             opt_cam.take_camera_frame, tolerance)
7         target = target + period  # sets next time independent of the body
8     time.sleep(.001)
```

Listing 10: Sample solution for periodicity.

The buggy implementation sleeps for at least two seconds after running the body of the **`while`** loop. However, it fails to take into account the time it takes to execute the body (Lines 3-4). A correct implementation can be seen in Listing 10. This implementation keeps track of time independently of how long the body runs. It does so by storing the next time to iterate from the start of the periodic loop (on Line 2) and modifying it by the period (on Line 7). After completing its current iteration, it polls repeatedly on Line 4 when the next time occurs and busy waits on Line 8. An alternative solution would be to sleep until the next iteration starts to avoid busy waiting. This would work by replacing Line 8 with `time.sleep(target - time.time())` and removing Line 4. Although `time.sleep` does not accept negative numbers, the code requirements specify not to worry about this case, which happens when the body takes longer than the period to execute.

All of our participants spent the majority of the time writing the vanilla Python solution. There is unavoidable code repetition to set up periodic control loops for the child and main processes that contribute

to this. Two of our four participants produced buggy solutions. Although P3 was unable to create a nondrifting periodic loop in the first part of the task for the asynchronous data generator, they rectified their misunderstanding when implementing the periodic loop for the parent process running the main loop. Their mistake was forgetting to call `start()` on the asynchronous process after defining it. Although this was a simple mistake, debugging this is very difficult. If the child process does not start, it does not put any data into the queue for the parent process. From the parent process's perspective, this shows as no data going into the queue. The participant needs to identify that the error's root cause stems from the child process side and then continue debugging from there. P4 was unable to figure out how to accurately create a periodic loop that did not drift over time for the child process. This led to them repeating the same issue in the second part for creating a periodic loop in the parent main process.

4.3 Task: Data Synchronization (DS)

In this task,[2] participants write code synchronize the asynchronous data streams created in the prior task. The definition of synchronizing data for a given function is that each function call's arguments have time intervals that overlap for each pairwise comparison of the arguments' intervals. For example, in the SI app, we call `local_fusion` with an image and LIDAR data. The image and LIDAR data should be generated around the same time (say, around 1:01) to prevent stale data from being used. If an image was timestamped from 1:00-1:02 and LIDAR data was timestamped from 1:01-1:03, they can be used together for a synchronized call for `local_fusion`. A visual explanation of this interval overlap between a pair of data is seen in Figure 4.4.

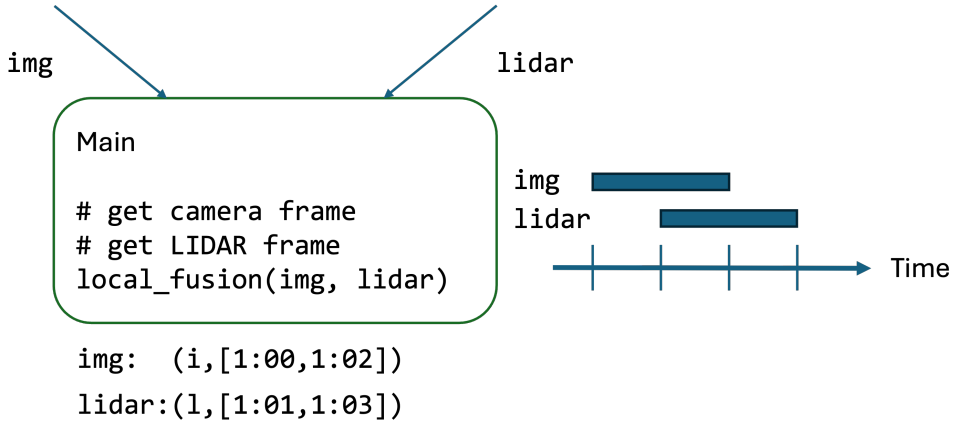


Figure 4.4: `local_fusion` requires a synchronized `img` and `lidar` before executing.

In the smart intersection, the participant synchronizes data between the camera and LIDAR sensors for the function `local_fusion`. In the urban flooding network, the participant synchronizes data between an optical image, GPS, and IMU to create metadata associated with the image using the `add_exif` function.

[2]Refer to A.1.4 to see the instructions given and the starter code.

4.3.1 TTPython Implementation

TTPython provides data synchronization for free in its main GRAPHified function. SQs need to satisfy their firing rule before they can execute their encapsulated function. The default Data Validity firing rule satisfies the Data Synchronization Task. What remains is for the participant to understand the SQ DV firing rule and to call the function with the correct arguments.

```
1  @SQify
2  def add_exif(opt_img, gps_exif, imu_data):
3      ...
4
5  @GRAPHify
6  def main(trigger):
7      with TTClock.root() as root_clock:
8          ...
9          opt_image = take_opt_image(sample_window,
10                                    TTClock=root_clock,
11                                    TTPeriod=2_000_000,
12                                    TTPhase=0,
13                                    TTDataIntervalWidth=500_000)
14
15         gps_data = take_gps(sample_window,
16                             TTClock=root_clock,
17                             TTPeriod=2_000_000,
18                             TTPhase=0,
19                             TTDataIntervalWidth=500_000)
20
21         imu_data = take_imu(sample_window,
22                             TTClock=root_clock,
23                             TTPeriod=2_000_000,
24                             TTPhase=0,
25                             TTDataIntervalWidth=500_000)
26
27         ###### T Task 2: Call `add_exif` with `opt_image`, `gps_data`, and
28         # `imu_data`. Ensure that these values are synchronized, or that their
29         # data intervals overlap in time.
30         exif_img = add_exif(opt_image, gps_data, imu_data)
```

Listing 11: The TTPython solution for the Data Sync Task in the UF app. The highlighted sections show the additions for a sample solution.

This task structure is quite similar to the ADG Task. Once again, the participant modifies two sections of the code. First, they add the `@SQify` decorator on Line 1 in Listing 11 to make the function visible to the TTPython compiler. They then use the SQified function `add_exif` on Line 30 to insert the SQ into the compiled graph.

4.3.2 Python Implementation

Once again, participants need to write code with low-level timing libraries. The vanilla Python implementation would need users to both design a relationship between data and time and to synchronize data from

their timestamps. The former can be challenging to design, so we provide scaffolding in the form of a library to separate data type design from synchronization implementation. A class named `TimedData` is provided to pair arbitrary data with time interval (`Interval`) objects. These classes were introduced in the tutorial with sample use cases provided. This scaffolding allows the user study design to abstract the main goal of the task (synchronizing heterogeneous, asynchronously generated data) through developing code to compare timestamps. The design abstracts the different asynchronous data streams (*i.e.* camera, LIDAR sensor) as sets that accumulate data over time. This design prevents participants from needing to rewrite code from the ADG Task. Now, the participant only needs to write code to synchronize `TimedData` objects within a set. The task is broken down into three parts: finding overlap between two time intervals, removing a candidate synchronized pair or trio of data across sets, and using the two steps above to find synchronized data arguments before calling the specified function with it. Finding if intervals had a non-zero intersection was intentionally left open-ended for participants to write. As the set interface design was predetermined, we provided clear scaffolding on how to implement set modification. Otherwise, programmers would have to write more code interfacing with ITP queues and processes.

4.3.3 Observations

In TTPython, P2 forgot to add the `@SQify` decorator, which the TTPython compiler promptly rejected. This requirement to have all function calls needing a TT decorator (`@SQify` or `@STREAMify`) was quizzed in the TTPython tutorial. The compiler error stating how it failed to find a SQified or STREAMified function was enough to clue P2 to add the `@SQify` decorator. P2 and P4 remarked how TTPython's SQ abstraction automatically synchronized data before function calls, indicating their understanding of the SQ's DV firing rule.

In vanilla Python, one participant had a bug with a corner case by being too restrictive for intervals that did overlap. To clarify how close their implementation is (found in Listing 12), we compare it to a sample solution in Listing 13.

```
def overlapping_timestamps(interval_x: Interval,
                           interval_y: Interval) -> bool:
    # interval_x.start < interval_y.end
    if interval_x.to_list()[0] < interval_y.to_list()[1]:
        # interval_x.start > interval_y.start
        if interval_x.to_list()[0] > interval_y.to_list()[0]:
            return True
    # interval_y.start < interval_x.end
    if interval_y.to_list()[0] < interval_x.to_list()[1]:
        # interval_y.start > interval_x.start
        if interval_y.to_list()[0] > interval_x.to_list()[0]:
            return True
    return False
```

Listing 12: A buggy vanilla Python solution for checking overlapping intervals.

```
def overlapping_timestamps(interval_x: Interval,
                           interval_y: Interval) -> bool:
    return not(interval_x.end < interval_y.start or
               interval_y.end < interval_x.start)
```

Listing 13: A sample vanilla Python solution for checking overlapping intervals.

If we invert each clause of the solution's return expression in Listing 13, we see that the cases are similar to the top-level `if` statements in the buggy implementation. However, the strict less-than comparison on Lines 4 and 9 incorrectly excludes the case when `interval_x` and `interval_y` are the same interval. Changing these lines is not enough, as Lines 6 and 11 fail the same case.

P4 had some difficulty understanding the custom `TimedData` library API. The first part of the task is to implement `get_overlapping_data` that compares between `Interval` classes to test if there is an overlap. The second part of the task takes sets of `TimedData` (that contain `Interval`) and calls `overlapping_timestamps` on those `Interval` fields. P4's implementation was calling `overlapping_timestamps` on the `TimedData` objects rather than their `Interval` fields. We attribute this fault to Python's dynamic typing, as we did not see this issue in other participants.

4.4 Task: Networking (NTWK)

In this task,[3] participants complete an application and write the networking code to send data across devices.

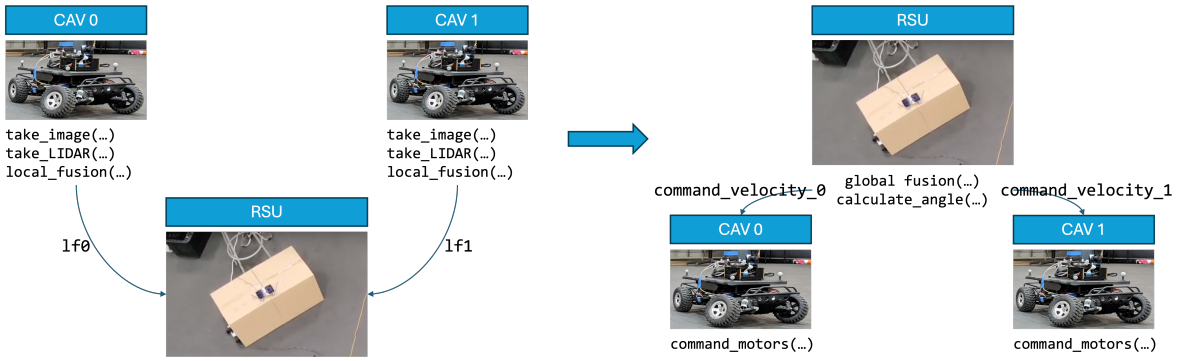


Figure 4.5: Architectural description of the Smart Intersection application. The CAVs first send `local_fusion` to the RSU, which then replies with `global_fusion` data.

In the smart intersection, the CAVs first generate LIDAR and camera information and synchronize them in `local_fusion`. It sends this to the RSU. The RSU calls `global_fusion` with info from both CAVs. It plans each CAV's routing with `calculate_angle` and sends it to each.

In the urban flooding network, the optical camera device combines an optical image with the GPS and IMU data in `add_exif`. It sends this to a thermal camera device, which calls `coregistration` with it and a thermal image. This is then sent to the device `"router"` that calls `has_flooding` on it and `upload_status`.

4.4.1 TTPython Implementation

TTPython uses a *macroprogramming* paradigm where the user declaratively specifies code assignments across devices in the network. The participant uses the `TTConstraint` construct in TTPython to assign

[3]Refer to A.1.5 to see the instructions given and the starter code.

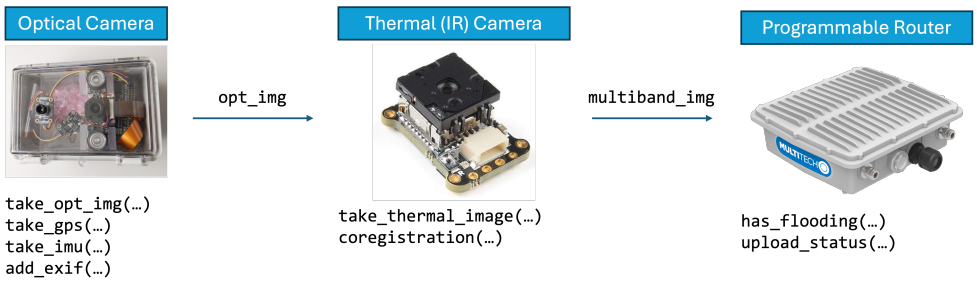


Figure 4.6: Architectural description of the Urban Flooding Network application.

SQs within its block to named devices. As TTPython will automatically assign SQs that are not specified under `TTConstraint` to a default device, there is no good indication to the user that the SQ has been assigned to that device without exposing implementation details. To avoid this, the instructions specify that all SQs in the `TODO` block must be assigned to a device.

```
@GRAPHify
def main(trigger):
    with TTClock.root() as root_clock:
        ...

        ###### T Task 3:
        # NOTE: All SQs below should be assigned to a device.
        # TODO: assign `take_opt_image`, `take_gps`, `take_imu`, and
        # `add_exif` to the device named "opt_camera".
        # TODO: Assign `take_thermal_image` and `coregistration` to device
        # named "ir_camera"
        with TTConstraint(name="opt_camera"):
            images = take_opt_image(sample_window,
                                    TTClock=root_clock,
                                    TTPeriod=2_000_000,
                                    TTPhase=0,
                                    TTDataIntervalWidth=500_000)
            gps_exif = take_gps(sample_window,
                                TTClock=root_clock,
                                TTPeriod=2_000_000,
                                TTPhase=0,
                                TTDataIntervalWidth=500_000)
            imu_data = take_imu(sample_window,
                                TTClock=root_clock,
                                TTPeriod=2_000_000,
                                TTPhase=0,
                                TTDataIntervalWidth=500_000)
            exif_img = add_exif(images, gps_exif, imu_data)

        with TTConstraint(name="ir_camera"):
            lepton = take_thermal_image(sample_window,
                                        TTClock=root_clock,
                                        TTPeriod=2_000_000,
                                        TTPhase=0,
```

```
                                                TTDataIntervalWidth=500_000)
            multiband_image = coregistration(exif_img, lepton)

        # TODO: Assign `has_flooding` and `upload_status` to the device named
        # "router".
        with TTConstraint(name="router"):
            classify = has_flooding(multiband_image)
            uploaded = upload_status(classify)
```

Listing 14: A sample TTPython solution for the UF app. The highlighted sections of code show the delta between what is presented as starter code and the solution.

4.4.2 Python Implementation

In the vanilla Python code, the participant uses RabbitMQ as the message broker to handle network connections. Unfortunately, RabbitMQ also uses the *queue* terminology as its network abstraction. We could not avoid the overloading of the queue concept between RabbitMQ and the `Queue` class in concurrency. To distinguish between this and interprocess queues used for concurrency, we specify that RabbitMQ queues are *network* queues while interprocess queues are *ITP* queues. For each network connection in the application, the participant must do the following for the sending and receiving devices.

- Initialize the RabbitMQ connection on the sender.
- Send data through the network queue on the sender.
- Set up a child process on the receiving device to listen on the network queue.

The 3rd item is implemented once with `listen_for_input`, which the participant writes. Its high-level implementation is shown below in Figure 4.7.

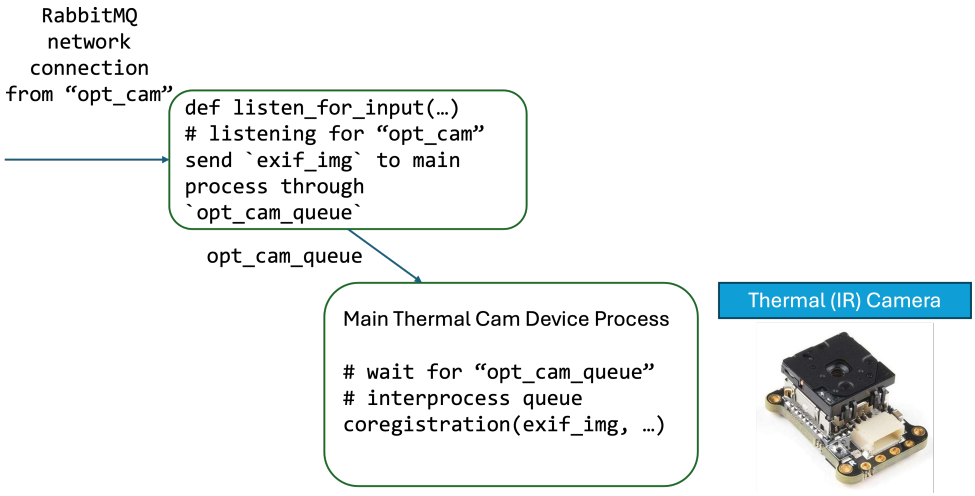


Figure 4.7: Architecture design for receiving an optical image with EXIF data from the optical camera to the thermal camera in the urban flooding network application.

This implementation closely follows the `send.py` code found in the RabbitMQ section in the vanilla Python tutorial. The difference is the abstraction to work with differently named network queues and ITP queues. The TODOs that the participant follows are exact as they describe initialization code required to set up the receiving network connection and how to create a callback function to receive messages. The participant implements a RabbitMQ network queue with given name that listens with separate process, as reading from a network queue is a blocking operation. A child process running `listen_for_input` does this and abstracts the network receiving as an asynchronous data generation source. The user can then interface with this as Python set, similar how the task in Section 4.3 was presented.

4.4.3 Observations

An unintentional design in the UF TTPython variant was to present the SQs out of order. The presented order is seen in Listing 15.

```
# TODO: assign `take_opt_image`, `take_gps`, `take_imu`, and
# `add_exif` to the device named "opt_camera".
# TODO: Assign `take_thermal_image` and `coregistration` to device
# named "ir_camera"
lepton = take_thermal_image(...)
images = take_opt_image(...)
gps_exif = take_gps(...)
imu_data = take_imu(...)
exif_img = add_exif(...)
multiband_image = coregistration(...)
```

Listing 15: An out-of-order presentation of SQs for labeling in the TTPython UF Task NTWK. The first and last SQ in the order are to be assigned to the `"ir_camera"` device.

The SQs `take_thermal_image` and `coregistration` should be assigned to the `"ir_camera"` device. P1 and P2 presented two different valid solutions for TTPython in Listings 16 and 17, respectively.

```
with TTConstraint(name="opt_camera"):
    images = take_opt_image(...)
    gps_exif = take_gps(...)
    imu_data = take_imu(...)
with TTConstraint(name="ir_camera"):
    lepton = take_thermal_image(...)
    multiband_image = coregistration(...)
```

Listing 16: P1's solution for TTPython's UF NTWK Task. The highlighted line shows the visual difference between the solutions.

```
with TTConstraint(name="ir_camera"):
    lepton = take_thermal_image(...)
with TTConstraint(name="opt_camera"):
```

```
    images = take_opt_image(...)
    gps_exif = take_gps(...)
    imu_data = take_imu(...)
with TTConstraint(name="ir_camera"):
    multiband_image = coregistration(...)
```

Listing 17: P2's solution for TTPython's UF NTWK Task. The highlighted line shows the visual difference between the solutions.

P1 moved the `take_thermal_image` SQ and grouped it with the `coregistration` SQ. This move does not change the program's semantics because the movement does not change the graph construction (as no data dependencies are modified). The `TTConstraint` construct only labels the SQs, and the runtime manager for TTPython assigns the labeled SQs to the named devices that subscribed before the program begins execution. This is what allows P2's solution to also be valid. P2 did not move the `take_thermal_image` SQ and instead inserted a new `TTConstraint` for it.

Our participants faced numerous challenges in the vanilla Python NTWK task. P1 stated their confusion about the initialization of sending and receiving code on the CAV for the SI app. They had to be reminded about the network architecture in Figure 4.5 to understand the difference between the constructors for network queues and ITP queues. Implementing `listen_for_input` (as seen in Listing 18) was challenging for all four of our participants.

```
def listen_for_input(rabbitmq_queue_name, interprocess_queue: mp.Queue):
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()
    channel.queue_declare(queue=rabbitmq_queue_name)

    def callback(ch, method, properties, body):
        interprocess_queue.put(decode_data_bytes(body))

    channel.basic_consume(queue=rabbitmq_queue_name,
                          on_message_callback=callback,
                          auto_ack=True)
    channel.start_consuming()
```

Listing 18: A sample solution for `listen_for_input`. This code represents the receiving capabilities for a pair of devices.

P1, P2, and P4 initially put in a constant string instead of the `rabbitmq_queue_name` parameter for their queue to listen, but later fixed this when they supplied the arguments when creating their child process. P3 was initially confused about what to put in the `callback` function and attempted to publish data through the channel, when the callback here instead specifies that it had received data from that specific channel. They needed clarification that the `rabbitmq_queue_name` corresponded to the named RabbitMQ network queue, while the `interprocess_queue` was the queue connecting between the child and the main process for that device. P3 then forgot to add the default exchange for the sending code, but had no other issues after.

P2 ultimately timed out at 25 minutes after being very confused about the RabbitMQ architecture for the SI app. They were confused about the extra process queue from the CAV side, as they forgot that

each receiver needed an ITP queue to send its information back to the parent process. They also ran into naming issues for their network queues when they realized they needed separate queues to send and receive between the RSU and each CAV. This culminated in accidentally using the same network queue name for sending and receiving on each CAV, which led to unexpected behavior.

4.5 Task: Time-Triggered Exception Handling (TTEH)

In this task,[4] participants create time-triggered exception handling code when a deadline is missed. Because we have designed robust data synchronization before execution with the Task DS, the code will not run if one of the synchronized data sources becomes unresponsive. This robustness can be further improved by detecting when actions are late. For example, if a device is waiting for upstream data for too long (thereby passing a deadline), we can run exception handling code. This exceptional handling in response to a deadline appears in TTPython with the `TTFinishByOtherwise` construct with TT keywords or in vanilla Python with a watchdog timer. We first discuss why this situation can occur for each application.

In the smart intersection, the CAVs send their `local_fusion` data to the RSU over the network. Networks are unreliable, so the device code needs to be resilient to external failures. Let's say the connection to `"cav0"` is unreliable while `"cav1"` is always available. Although `global_fusion` ideally should have information from CAVs before making a decision, it can still make assumptions from the last time it heard from `"cav0"` and with `"cav1"`'s current position. `global_fusion` could still run even if a CAV becomes unresponsive for a while. The `"rsu"` could run time-triggered exception handling code to still run `global_fusion` knowing that `"cav0"` was unresponsive. In this task, participants write code that calls `global_fusion(missing_fusion_input(), lf1)` (where `lf1` is `"cav1"`'s `local_fusion`) when `"cav0"` becomes unresponsive. Figure 4.8 explains the application's control logic for the SI app's TTEH code.

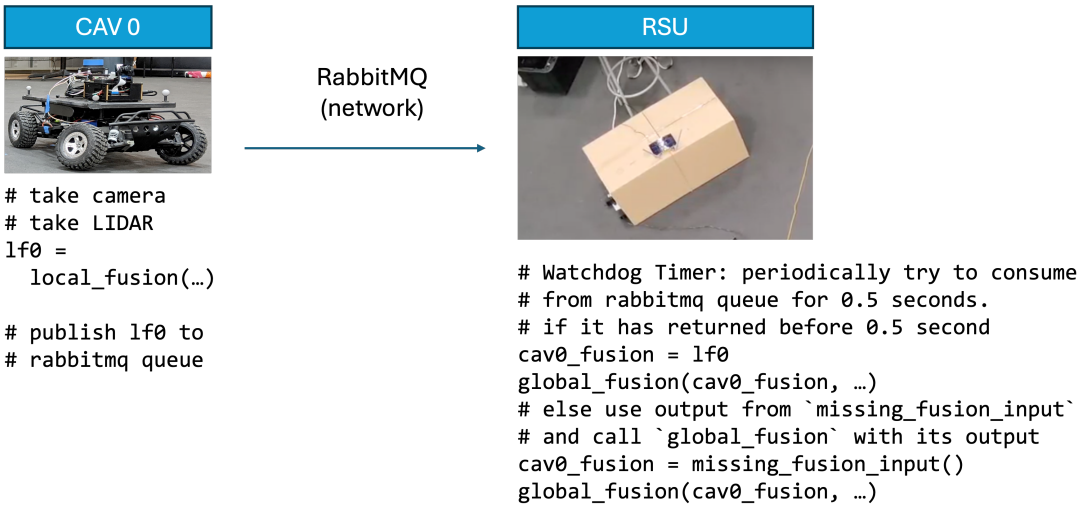
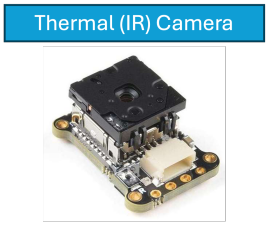


Figure 4.8: Starting instructions for vanilla Python implementation for the smart intersection.

In the urban flooding network application, the thermal camera takes an image and combines it with an optical image to create a multiband image. It sends this multiband image to a machine learning model

[4]Refer to A.1.6 to see the instructions given and the starter code.

to detect flooding. Adding infrared bands in the classified image gives more accurate information, as the visible spectrum information can vary depending on lighting conditions and color monotonicity. The hardware in this application is cost-effective and can have unexpected hardware failures. Specifically, the thermal camera (Flir Lepton 3.5) is flaky and can become unresponsive [34]. A power cycle (turning the camera off and on again) is used to reset the camera. Participants call the `reset_cam` function if the thermal camera is unresponsive and does not produce an image by the deadline. Figure 4.9 explains the application's control logic for the UF app's TTEH code.




```
# Plan B: periodically wait for thermal
# img for 1 second.
success_img = take_thermal_img()
# if it has returned before 1 second
coregistration(exif_img, …)
# else run `reset_cam` and call coreg with
# its value
replaced_val = reset_cam()
coregistration(replaced_val, …)
```

Figure 4.9: Starting instructions for TTPython implementation for the urban flooding network. TTPython uses the terminology *Plan B* for time-triggered exception handling.

4.5.1 TTPython Implementation

TTPython uses the `TTFinishByOtherwise` construct to implement time-triggered exception handling. The participant writes in the corresponding TT keywords to specify the data that it is waiting for, the deadline with `TTTimeDeadline`, the Plan B function (time-triggered exception handler) with `TTPlanB`, and a flag with `TTWillContinue` to specify whether to continue execution after executing Plan B. This is similar to a **`try`**/**`except`** exception handler, with either handling or raising the deadline time violation. If `TTWillContinue` is **`True`**, the execution will continue substituting the output with the output from Plan B. Otherwise, it will stop, and downstream SQs will be unable to fire due to missing the token necessary for synchronization. The participant also modifies the downstream SQs to use the `TTFinishByOtherwise` output. This is seen in Listing 19 on Line 29. The `coregistration` SQ is dependent on `thermal_img`. When `thermal_img` is replaced by the TTEH's output with `successful_img`, `coregistration`should be switched to use `successful_img` as well.

```
1 @GRAPHify
2 def main(trigger):
3     with TTClock.root() as root_clock:
4         ...
5
6         with TTConstraint(name="ir_camera"):
7             t_img_start_time, thermal_img = take_buggy_image(
```

```
                sample_window,
                TTClock=root_clock,
                TTPeriod=2_000_000,
                TTPhase=0,
                TTDataIntervalWidth=500_000)

            ###### T Task 4:
            # TODO: Call the Plan B `reset_cam()` for `thermal_img` before
            # `coregistration` if function `take_buggy_image` does not return
            # within `deadline_time`. Make sure that `coregistration` still
            # runs if Plan B fires.
            deadline_time = READ_TTCLOCK(t_img_start_time,
                                         TTClock=root_clock) + 1_000_000
            successful_img = TTFinishByOtherwise(thermal_img,
                                                 TTTimeDeadline=deadline,
                                                 TTPlanB=reset_cam(),
                                                 TTWillContinue=True)

            # TODO: replace `thermal_img` with output of the Plan B handler.
            # Make sure that `coregistration` still runs if Plan B fires.
            # the starter code is shown below
            # multiband_image = coregistration(exif_img, thermal_img)
            multiband_image = coregistration(exif_img, successful_img)

        ...
```

Listing 19: A sample solution for the TTPython UF TTEH Task. The highlighted sections show the additions for a sample solution. Line 29 is shown to show the difference between the starter code and the solution.

4.5.2 Python Implementation

We see a sample solution of the SI application in Figure 20.

```
def rsu_main(start_time):
    ...

    # periodic control loop for the CAV
    while True:
        curr_time = time.time()
        if next_time <= curr_time:
            deadline_time = curr_time + deadline_offset

            ###### T Task 4
            # NOTE: This is the normal path if everything comes in on time.
            # You don't need to do anything here.
            success = False
            lf0_came = False
            lf1_came = False
            while time.time() < deadline_time and not success:
```

```
17                 ...
18
19             ###### T Task 4
20             # This is the exceptional handling time. This should occur if
21             # the "rsu" fails to synchronize data between both CAVs.
22             if not success and not lf0_came:
23                 cav0_fusion = missing_fusion_input(next_time)
24
25                 # find cav1 match
26                 cav1_fusion = extract_fusion(cav1_fusions,
27                                              cav0_fusion.interval)
28
29                 if cav1_fusion is not None:
30                     interval = cav1_fusion.interval
31
32                     gf = global_fusion(cav0_fusion.data, cav1_fusion.data)
33                     command_velocity_0 = calculate_angle(
34                         gf, cav0_fusion.data, cav_0)
35                     command_velocity_1 = calculate_angle(
36                         gf, cav1_fusion.data, cav_1)
37                     cv0 = TimedData(command_velocity_0,
38                                     interval.to_list())
39                     cv1 = TimedData(command_velocity_1,
40                                     interval.to_list())
41                     channel0.basic_publish(exchange='',
42                                            routing_key=queue_name0,
43                                            body=encode_data(cv0))
44                     channel1.basic_publish(exchange='',
45                                            routing_key=queue_name1,
46                                            body=encode_data(cv1))
47             ...
```

Listing 20: A sample solution for the vanilla Python UF TTEH Task. The highlighted sections show the additions for a sample solution.

To reduce the difficulty of creating a watchdog timer, the participant does not have to worry about writing timing code to generate a timer. They are guided by a series of TODOs that abstract time-triggered exception handling by checking boolean flag changes. These TODOs are direct as there is a lot of changes between using the `TimedData` classes and its field data. The participant writes these boolean flags on Line 22. The participant needs to recreate the functional path of missing a deadline and sending the output of the time-triggered exception handler code over the network. We do require participants to unpack data from our custom class `TimedData`, as we assume that the underlying libraries and code operate on untimed data. This unpacking is seen on Line 34 for `cav0_fusion`. Although the Python implementation requires less of an understanding of the design of the system, the participant has more code to write and has to ensure that the object types are correct with the functions they use.

4.5.3 Observations

For the SI TTPython Task, P3 originally set `TTWillContinue=False` when instead it should have been set as `True`. They later commented on how they misread what the option does. We believe that this

flag option is more difficult to understand than similar options in `@STREAMify` as it specifies a specific change in the control plane of the dataflow graph. Most TT options in TTPython are static parameters dependent on the environment, such as `TTPeriod` and `TTDataIntervalWidth`. P3 and P4 wanted to explore the semantics of `TTFinishByOtherwise` by adding extra Python code. P3 attempted to write control code to check for missed deadlines, similar to how the vanilla Python implementation does in its corresponding TTEH task. The control code includes **`if`** statements that are not implemented in TTPython's dataflow graph structure. This would still be difficult even if TTPython supported **`if`** statements, as this approach would lack the preemptive interrupt required to implement a deadline timing decision. P4 tried to add `print` statements in the GRAPHified main body, but the TTPython compiler failed to find a SQified `print` function.

In vanilla Python, P1, P2, and P4 expressed confusion on what data type to provide to the function running after the exception handler (such as the call to `global_fusion` on Line 32). A quick check of the function's signature, coupled with the TODO, clarified some of the confusion. This was a recurring pattern in the vanilla Python implementation for the CE Task, which has similar type requirements for its application-specific functions.

4.6 Task: Code Evolution (CE)

In this task[5], participants take a completed application from the additions in the previous two tasks and refactor the code to run on different devices and/or use different time-triggered exception handling code. The prior tasks have been designed to be as parallel as possible, but the evolution task is tailored to each application. Thus, we explain each application with its TTPython and Python implementation separately.

4.6.1 Smart Intersection

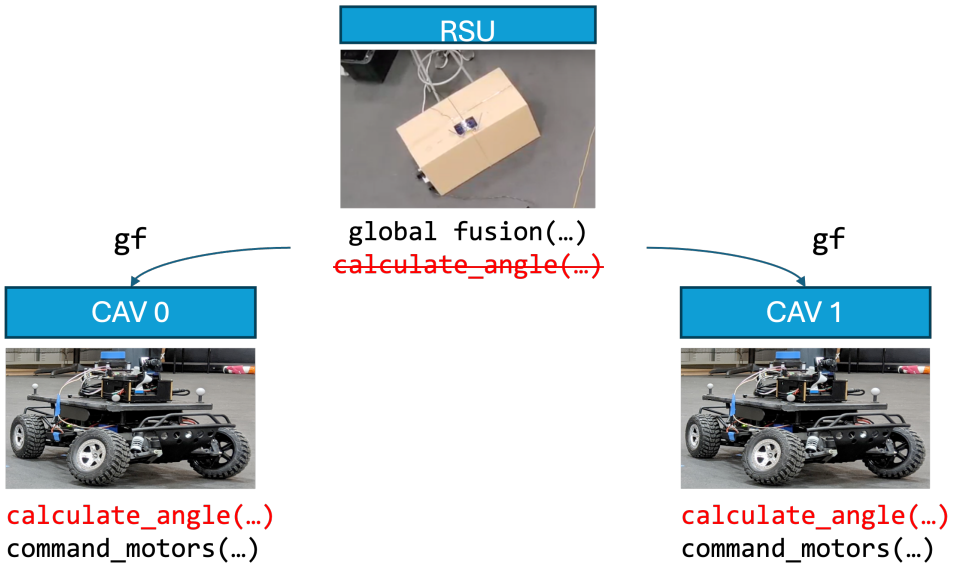


Figure 4.10: Visualizes the code movement of `calculate_angle` from the RSU to each CAV.

[5]Refer to A.1.7 to see the instructions given and the starter code.

The high-level architecture is repeated below, and a visual representation can be seen in Figure 4.5. The CAVs first generate LIDAR and camera information and synchronize them in `local_fusion`. It sends this to the RSU. The RSU calls `global_fusion` with info from both CAVs. It plans routing for each CAV with `calculate_angle` and sends it to each CAV, which uses this information in `command_motors`.

The current architecture described is overly restrictive for the CAV's autonomy. In practice, the RSU is only used to make decisions when CAVs arrive at the intersection to resolve ambiguity on which car has priority for moving. To reflect this architectural change, the route calculation done in `calculate_angle` can be done locally on each CAV. This can be used with a time-triggered exception handler to only use the CAV's `local_fusion` output if the CAV fails to hear from the RSU's `global_fusion` in time. In this application, the participant moves the code for `calculate_angle` onto each CAV and changes the corresponding time-triggered exception handler. The overall flow of data is unchanged with this new application: the CAVs still produce sensor information locally, send this to the RSU, which responds to the CAVs with global information. The challenge in the smart intersection code evolution task is to keep track of the type of data being sent over the network and change the functional code in response.

TTPython Implementation

The pertinent portion of the program to modify is shown in Listing 21.

```
###### T Task 5
with TTConstraint(name="rsu"):
    ...
    gf = global_fusion(cav_0_fusion, cav_1_fusion)
    # TODO: Move each `calculate_angle` for "cav0" and
    # "cav1" to run on its respective device "cav0" and "cav1".
    command_velocity_0 = calculate_angle(gf, lf0, cav_0)
    command_velocity_1 = calculate_angle(gf, lf1, cav_1)

with TTConstraint(name="cav0"):
    velocity_0 = TTFinishByOtherwise(command_velocity_0,
                                     TTTimeDeadline=cav_0_deadline,
                                     TTPlanB=emergency_stop(),
                                     TTWillContinue=True)
    final_result_0 = command_motors(velocity_0, cav_0)

with TTConstraint(name="cav1"):
    velocity_1 = TTFinishByOtherwise(command_velocity_1,
                                     TTTimeDeadline=cav_1_deadline,
                                     TTPlanB=emergency_stop(),
                                     TTWillContinue=True)
    final_result_1 = command_motors(velocity_1, cav_1)
```

Listing 21: SI TTPython CE Task starter code description

```
###### T Task 5
with TTConstraint(name="rsu"):
```

```
3       ...
4       gf = global_fusion(cav_0_fusion, cav_1_fusion)
5
6   with TTConstraint(name="cav0"):
7       safe_gf0 = TTFinishByOtherwise(gf,
8                                      TTTimeDeadline=cav_0_deadline,
9                                      TTPlanB=no_global_fusion(),
10                                     TTWillContinue=True)
11      velocity_0 = calculate_angle(gf0, lf0, cav_0)
12      final_result_0 = command_motors(velocity_0, cav_0)
13
14  with TTConstraint(name="cav1"):
15      safe_gf1 = TTFinishByOtherwise(gf,
16                                     TTTimeDeadline=cav_1_deadline,
17                                     TTPlanB=no_global_fusion(),
18                                     TTWillContinue=True)
19      velocity_1 = calculate_angle(gf1, lf1, cav_1)
20      final_result_1 = command_motors(velocity_1, cav_1)
```

Listing 22: TTPython CE Task sample solution. The code is highlighted to emphasize the changes between 21.

`calculate_angle` in Listing 21 on Lines 7 and 8 needs to be moved **after** the `TTFinishByOtherwise` for each respective CAV, as seen in Listing 22 on Lines 11 and 19. `TTFinishByOtherwise` now checks if `gf` arrives over the network in time, so any downstream SQs using `gf` on the CAVs need to be modified as well. `calculate_angle`'s `gf` argument then needs to be updated to the output of `TTFinshByOtherwise` in Lines 7 and 15, which is coined as `safe_gf` in the solution Listing. Its `TTPlanB` (exception handler) is also updated to `no_global_fusion` instead.

Python Implementation

We first discuss the changes on the RSU before talking about the code changes in the CAVs[6]. The participant first moves the code for `calculate_angle` to each respective CAV. The data being sent over the network queue is changed to the output from `global_fusion`. The `global_fusion` data also needs to be synchronized with the particular periodic iteration before it can be used, which requires users to remember the interface they designed in Task DS. For each respective CAV file, the participant changes the local variable names responsible for handling the network data. We believe this is necessary as the code should accurately reflect the data type being sent to avoid future misconceptions. The difficulty in the vanilla Python task is finding all the locations to change the pertinent code while working between different files.

Observations

P3 and P4 completed the TTPython implementation. Both P3 and P4 struggled with understanding the architectural changes necessary for moving `calculate_angle` from the RSU to the CAV. The starter code has `TTFinishByOtherwise` checking whether sending `command_velocity` from `calculate_angle` was successfully sent over the network. When `calculate_angle` is moved

[6]Refer to A.1.7 to see the starter code.

to the CAVs, the data being sent over the network changes to the RSU's global fusion data (`gf`). Therefore, the TTPython solution requires not only code movement across `TTConstraint` **`with`** blocks but also `TTFinishByOtherwise` modifications. This requires a solid understanding of the application architecture. P3 had difficulty understanding that they needed to change the `gf` parameter for `calculate_angle` to `safe_gf`, the output of the updated `TTFinishByOtherwise`. P4 initially placed `calculate_angle` before its `TTFinishByOtherwise` Once they clarified their misconception of how `calculate_angle` interacts with the TTEH on the CAVs, they were able to complete the necessary changes.

P1 and P2 completed the Python implementation. P1 had issues with variable renaming to reflect the change of data from `command_velocity` to `gf` for each cav. P2 stated that it was difficult to know when to use the custom-timed data library for function calls. The `TimedData` library is necessary when data is being synchronized between different asynchronous data streams or sent over the network, but is not required when it is sequentially generated. For example, `command_motors` uses `calculate_angle`'s output (`command_velocity`) as its argument. In the original implementation, `command_velocity` was sent over the network and needed to be wrapped in a `TimedData` object as it could arrive out-of-order. However, once it is moved to the CAV, it no longer needs this encapsulation as the local execution is ordered. The `command_velocity` argument to `command_motors` then changes from `TimedData` to only the base data and no longer needs to be unwrapped.

4.6.2 Urban Flooding Network

The high-level architecture is repeated below, and a visual representation can be seen in Figure 4.6. The optical camera device takes an image and combines this with the GPS and IMU data in `add_exif`. It sends this to the thermal camera device, which calls `coregistration` with it and a thermal image. This is then sent to the device `"router"`, which calls `has_flooding` on it and `upload_status`.

In this task, we will move `coregistration` (on the thermal camera device) to the router, as it has better computing power. The challenge in the urban flooding network code evolution task is the change of network topology. The optical camera device no longer sends its optical image to the thermal camera device; instead, it sends it to the router, as it is where `coregistration` is being run. The thermal camera changes the data it sends to the router: from a multiband image to a thermal image.

TTPython Implementation

TTPython's macroprogramming approach makes this task trivial. Because data dependencies are symbolically linked at compile time and network routing is done at runtime, the network changes to implement this are given for free as long as the participant defines the variables correctly. A single line change is required to move `coregistration` on Line 4 in Listing 23 from the `TTConstraint(name="ir_camera")` block to the `TTConstraint(name="router")` block on Line 7.

```
1 with TTConstraint(name="ir_camera"):
2     ###### T Task 5:
3     # TODO: Move `coregistration` to the device "router".
4     # multiband_image = coregistration(exif_img, successful_img)
5
6 with TTConstraint(name="router"):
7     multiband_image = coregistration(exif_img, successful_img)
```

```
8     classify = has_flooding(multiband_image)
9     uploaded = upload_status(classify)
```

Listing 23: TTPython UF CE Task sample solution. The participant's starter code has Line 4 uncommented and does not have Line 7 included.

Python Implementation

The vanilla Python implementation is more involved in the architectural changes, as the participant has to handle networking requirements.[7] Firstly, the RabbitMQ connection from the optical camera to the thermal camera must be changed to go to the router instead. Secondly, the code to synchronize an optical image with a thermal image needs to be modified to work on the router. Most of the control logic can stay unchanged, but the participant needs to ensure that the overall application logic stays consistent over the refactor. Finally, the data type from the thermal camera being sent over the network changes. To streamline this, the participant comments and uncomments code responsible for unpacking and packing code in our `TimedData` custom class.

Observations

P1 and P2 completed the TTPython implementation. P1 seemed hesitant about their solution with a single line change. They confirmed their answer with the sample output and moved on. Interestingly, P2 was extremely confident in their solution and finished in record time, taking less than 2 minutes.

P3 and P4 completed the Python implementation. P4 had an interesting pause when considering the queue names for the RabbitMQ network queue linking. The movement of `coregistration` to the router meant that the optical camera changes its outbound device from the thermal camera to the router. The thermal camera was connected by the RabbitMQ queue name `'ir_to_router'`. P4 first changed the optical camera's queue name to `'router'`. P4 then saw the original connection between the thermal camera and the router, referred back to their optical camera code, and then changed the queue name for the optical camera to the `'ir_to_router'`, mistaking the network connection. They then realized their mistake after visiting the TODO to set up the `listen_for_input` on the router between the optical camera.

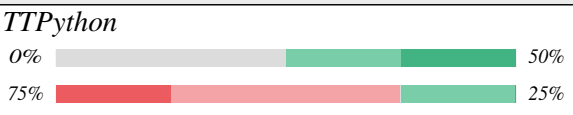
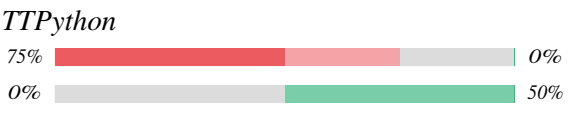
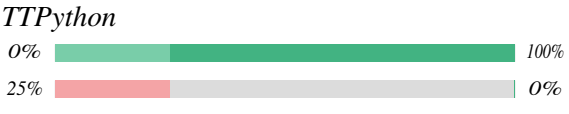
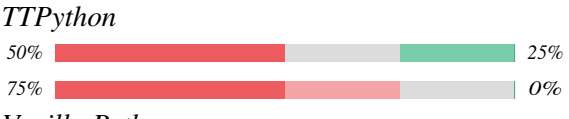
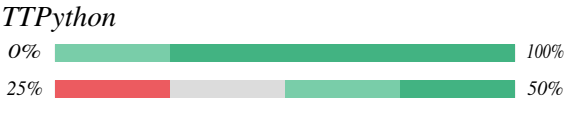
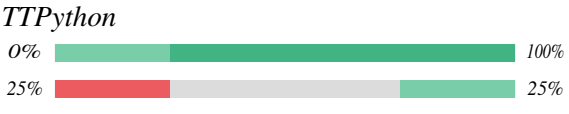
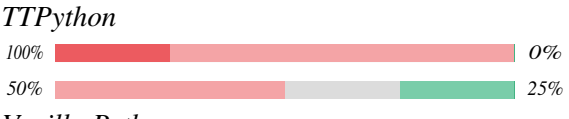
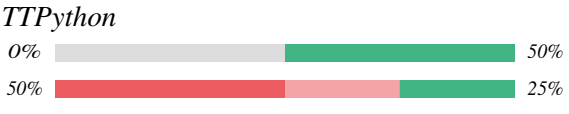
4.7 Results

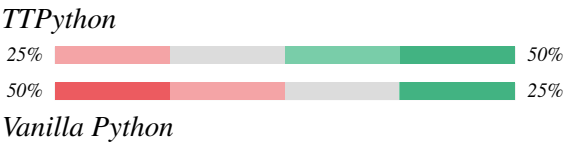
4.7.1 Post Study Questionnaire

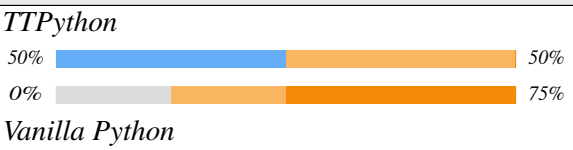
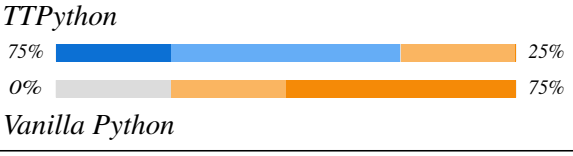
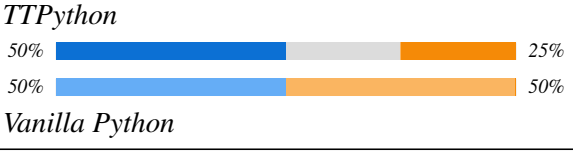
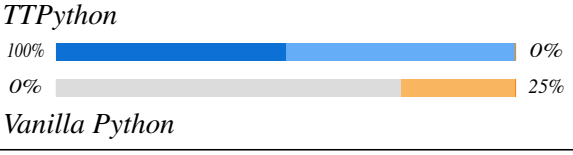
We ran the user study with 4 participants. After completing both applications, participants filled out a questionnaire on their experiences with each system. These questionnaires used a combination of System Usability Scale (SUS) [11] questions on a 5-point scale and the NASA Task Load Index (TLX) [28] on a 10-point scale. These questions collect data on perceived workload and system usability to help us understand the challenges participants faced using each system.

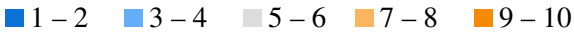
[7] Refer to A.1.7 to see the starter code

Table 4.1: An overview of the System Usability Scale [11] and the NASA Task Load Index (TLX) results [28].

Description	Distribution
System Usability Survey [11]	
I think that I would like to use this system frequently.	
I found the system unnecessarily complex.	
I thought the system was easy to use.	
I think that I would need the support of a technical person to be able to use this system.	
I found the various functions in this system were well-integrated.	
I thought there was too much inconsistency in this system.	
I would imagine that most people would learn to use this system very quickly.	
I found the system very cumbersome to use.	
I feel very confident using the system.	

I needed to learn a lot of things before I could get going with this system.	

Description	Distribution
NASA Task Load Index (TLX) [28]	
How mentally demanding was the task? *(1 - Very low, 10 - Very high)*	
How physically demanding was the task? *(1 - Very low, 10 - Very high)*	
How hurried or rushed was the pace of the task? *(1 - Very low, 10 - Very high)*	
How successful were you in accomplishing what you were asked to do? *(1 - Perfect, 10 - Failure)*	
How hard did you have to work to accomplish your level of performance? *(1 - Very low, 10 - Very high)*	



Looking at the SUS questions in Table 4.1, we see that TTPython does well in questions such as "*I thought the system was easy to use.*", "*I found the various functions in this system were well-integrated.*", and "*I found the system very cumbersome to use*". This trend in the data suggests that TTPython's abstractions do well describing our selected tasks. It suggests that DT applications have timing and distribution cross-cutting concerns for participants to favor TTPython. P4 said,

> What I'm balancing between [for TTPython] is having less custom stuff to work with versus having a good framework that takes away a lot of the complexities.

TTPython also does relatively well on self-reported task load with the NASA TLX questions. Participants were split on how mentally demanding TTPython was. We hypothesize this is caused by difficulty understanding TTPython's dataflow graph semantics. For example, P3 and P4 struggled more with TTPython's TTEH task as discussed in Section 4.5.3. Furthermore P4 said,

> It's like something that's almost sequential but also separate processes, so that's interesting

> to me. . . . It's something you have to spend a little bit of time to get used to, but once you get used to it it's pretty straightforward to use.

TTPython does very well in Q5 (*How hard did you have to work to accomplish your level of performance?*). We believe that TTPython scores well in this category due to the localization of the solution. Each TTPython variant of the tasks is much smaller (in lines of code) in comparison to their vanilla Python solution. P1 commented,

> For the first one [vanilla Python], while I was following the TODOs, I was trying to keep in my head everything I was doing, and at one point, I kind of got really confused, and I had to step back and think about everything really hard. . . . I felt a lot of mental load because there was just a lot more code. . . . There were a lot more little things that I could get wrong in any given place.

The declarative nature of TTPython coupled with the smaller amount of code edits required to specify or make architecture modifications contributes to the perceived difficulty. P2 said,

> . . . for the second one [TTPython], it was part of the framework itself. There's a decoupling of the computation that you're doing from where it's being run. You tell it this runs here and this one's here. That was easier to think about.

TTPython's orthogonality in expressing different timing and distribution requirements reduces the participants perceived cognitive load.

> TTPython was a lot more straightforward. It was like, this decorator does this, and this is how you call it. It was just a lot more straightforward.

4.8 Discussion

A common theme throughout the vanilla Python tasks was the amount of information participants had to keep track of while coding the solution. P1 remarked:

> For the first one [vanilla Python], while I was following the TODOs, I was trying to keep in my head everything I was doing, and at one point, I kind of got really confused, and I had to step back and think about everything really hard.

Each comparable vanilla Python task to its equivalent TTPython version has a larger amount of startup code required and can have varying cross-cutting concerns. We can see this example in both timing and distribution requirements.

For timing, P3's statement about Task DS captures the major differences between the TTPython and vanilla Python implementation.

> I did more numbers specification in TTPython, but I didn't have to do as much logical specification compared to [vanilla] Python . . .

For distribution, we saw how participants had trouble keeping track of their RabbitMQ network queue names. The separation between sending and receiving code in different files and devices is a major challenge. P3 states,

> With vanilla Python, if I had multiple networks to keep track of, that would be a lot more confusing. I have to keep track of the name of the channel and things moving around. Whereas TTPython, it's just tell it to [assign SQs to] the device, which [is] way simpler.

P2 had great difficulty with the vanilla Python NTWK task and timed out at 25 minutes. They mentioned how the virtualization of the CAVs and RSU on a single device was hard to keep track of when the source code for each device was next to each other. P2 said,

> Setting up all the listeners and senders and making sure that they're all coordinated [in RabbitMQ] is really difficult to think about and to organize.

We believe this highlights the advantage of TTPython's macroprogramming language design. In TTPython, the code assignment is declaratively specified at the SQ level, while vanilla Python implicitly specifies this at the file level. The user then does not have to worry about how to interface with network connections across files through a queue abstraction; instead, they are concerned with what data is used across different `TTConstraint` blocks.

One major challenge that participants faced when writing in vanilla Python was the amount of code necessary to reach a minimal amount of testing. Many of the vanilla Python tasks are difficult to debug because participants interact with different timing, networking, and concurrency code. TTPython's appearance to vanilla Python makes it easy enough for most participants to start writing immediately.

> If I were to choose one for the purposes of experimentation, I might go with the first one because it's easier to get started and easier to dip my toes in the water.

P3 remarked,

> ... all of the timing is handled in the backend. That's really nice because there's a lot less implementation on the front end. So I thought TTPython was easier because a lot of it [concurrency/timing implementation] was just abstracted away.

We had mixed reactions to the difficulty of understanding TTPython's TTEH Task, specifically for `TTFinishByOtherwise`'s semantics. P3 remarks,

> I was definitely confused by the exceptional time handling. With vanilla Python, if it goes over time, then send the "reset" and it's fine. TTPython felt more complicated ... [with having to] rewrite a bunch of the functions. It makes sense now, but it was hard to wrap my head around at first.

While P2 said,

> I would end up writing my own function for it, where the API looked exactly like TTPython['s] ... where you have how long you should wait and the callback. The TTPython API is exactly how that would look.

In the previous tasks, participants did not have to interact with the dataflow graph semantics of TTPython. The abstraction of TTPython decorators and `TTConstraint` in the **`with`** block works similarly to their imperative counterparts. However, `TTFinishByOtherwise` specifically works as edge modification in the dataflow graph. Our initial findings indicate that the application's architecture affects the understanding of how to use the time-triggered exception handler. We note that P3 did complete the TTPython tutorial much faster than other participants at around 22 minutes, while P2 took 53 minutes. Including pilot data, most participants finish the TTPython tutorial around 35-45 minutes.

4.8.1 Future Steps for TTPython

Although TTPython was favored among the four participants, the participants did highlight some negatives. P3 brought up a common complaint when visualizing dataflow graphs [56].

> The graphs were nice, but as soon as they got beyond the tutorial graphs, I didn't think they were going to be helpful. So I didn't even look at them. ... The lines were already overlapping so many times that it was hard to tell which labels went on which line.

Some participants were hesitant to use TTPython due to their unfamiliarity with dataflow graph execution semantics. P1 spoke about how the vanilla Python system still had merits in comparison to TTPython.

> I would want to know more details about how the timing is done [in TTPython] if I were to do systems stuff. Just so that I don't get caught in some sort of weird timing bug and to have more control over the system. The nice thing about the [vanilla Python] RabbitMQ example is that you have that control and you know where any potential errors are coming from. But it's also really annoying to deal with. I had a suboptimal solution the first time [for Task ADG] and then changed it to be more accurate but also more annoying [to implement].

P4 had a similar sentiment.

> It (vanilla Python) allows you to define more customized ... time management, whereas TTPython ... [is] more structured, and it's easier for someone to get into without much experience. With someone with more experience or with some more complex application, perhaps the customizability would be more desired.

One challenge for TTPython was creating a tutorial for the purposes of the user study. A tutorial had previously been designed to explain TTPython with a more holistic approach, teaching the intricacies of the dataflow graph semantics. We initially decided that such an approach was not appropriate for the user study due to time constraints. P1 spent the most time out of all other participants on the tutorial and spoke about the difficulty of the level of depth for the tutorial.

> In some ways, it's a little too abstract because I'm left with a lot more questions. I think having a lot of good examples or clear documentation that's written at different levels [of systems familiarity] of documentation would probably be really helpful.

We saw that misunderstanding TTPython's semantics led to issues when users tried to debug in the `@GRAPHify` body. P3 and P4 attempted to add Python code that would not pass the TTPython compiler due to not being implemented yet or calling functions that were not SQified. Instead, they had to move their debugging to within a SQ. This highlights a weakness in TTPython where it is difficult to add runtime debugging at the graph level. Because the `@GRAPHify` body textually represents a dataflow graph, using standard debugging procedures would introduce intrusive, system-wide changes. This raises questions on how to incorporate nonintrusive, system-level tools to debug program behavior for macroprogramming frameworks.

In general, the participants expressed how vanilla Python is more general and requires more effort to implement than TTPython, but is less structured. TTPython offers a tradeoff between the flexibility of a generic language and a structured, orthogonal framework for timing and distribution specifications.

4.9 Limitations and Threats to Validity

Most of our participants are college students and do not interact with distributed, time-sensitive applications. None of our participants specified experience working with message brokers in general. Additionally, although we required participants to have systems knowledge to ensure that they were familiar with processes, we did not require users to have prior experience using Python's `time` and `multiprocessing` libraries. The tutorial lengths between TTPython and vanilla Python are also different. We attribute this to needing to learn a new system and understanding dataflow graph semantics in TTPython. There are fewer concepts to understand in the vanilla Python implementation. Our sample size also makes it difficult to make any general claims about the system.

Although we randomized the order of the tool used by participants, we did not randomize the order in which the tasks were presented. More data needs to be collected to answer whether a learning effect was present.

Because the time library given for the vanilla Python implementation was custom-developed to mimic TTPython's synchronization capabilities, its design choices made it difficult to treat application functions as a black box. To minimize the impacts of this design choice, we explicitly specified in the instructions when to unpack or pack data with our custom library. This led to the vanilla Python tasks to become more mechanical than their TTPython counterparts, so it becomes unclear if participants reach the same understanding in the vanilla Python implementations. Prior pilots attempted to follow a high-level description like TTPython but found the difficulty unmanageable. The amount of code required by vanilla Python's implementation to replicate TTPython's capabilities also contributes to the difficulty. The TTEH Task used to include implementing the watchdog timer/timing code, but was cut down due to time constraints. Our results suggest that TTPython's structure allows participants to reason at a higher level than Python because it separates cross-cutting concerns.

We provided a sample environment for participants to run their code and compare their output against a sample solution for correctness. However, the tasks outside of vanilla Python's ADG and DS tasks are almost all-or-nothing, as these implementations cannot easily be incrementally tested. TTPython's tasks do not suffer as much when compared to their vanilla Python counterparts, as its search choice is more limited. The size of the code and executing in real-time contribute to this difficulty, so it becomes unclear whether providing a realistic environment helped participants understand the code.

4.10 Conclusion

DT applications are challenging to write and manage due to their cross-cutting concerns. TTPython leverages common timing and distribution concepts found in these applications, presenting them through a macroprogramming language and dataflow graph execution model. In this study, we examined difficulties when writing two realistic DT applications in TTPython and in vanilla Python with a message-broker system. We demonstrate that writing periodic code for asynchronous data streams, synchronizing data across them using time intervals and sets, and managing multiple RabbitMQ network queues is challenging in Python. We also find that mixing timing and distribution concepts can be difficult for TTPython users. Our preliminary results indicate that a timed, tagged-token dataflow graph model with a macroprogramming language shows promise as a learnable and usable abstraction for timing and distribution.

Chapter 5

Related Work

5.1 Distributed Systems and Time

Addressing distributed systems challenges has a rich history. X10 [12] is an object-oriented programming language targeting high-performance systems and introduced concurrency with partitioned global address spaces. It took Java as a base language and extended it to handle asynchronous activities in a distributed setting. Frameworks have also been language agnostic for handling generic computation. MapReduce [20] separated computation from the data it works on by presenting basic types such as sets and lists as abstractions for programmers to use while dealing with underlying system challenges of where data is stored and how to share computation. These large frameworks inspired work examining how to create better abstractions with distribution frameworks in mind, such as Spores [44]. It focused on designing functional code in a principled manner to avoid errors found in a distributed, concurrent environment. These bodies of work inspired the approach to memory management and code separation of SQs in TTPython.

Wireless Sensor Networks (WSN) [59] are often viewed as a precursor to modern cyber-physical systems, particularly those that rely on wireless communications. WSN research considered a wide variety of programming paradigms [47]. Node-centric programs may be written in low-level languages like C, in domain-specific variants like NesC[25], or as a set of high-level interpreted instructions as in Maté [37]. Popular embedded system languages and frameworks such as PRET-C [4] allow programmers to carefully manage resources on a device-level basis for small applications, but the mechanisms they provide prove unscalable when applications begin to expand over many devices. The underlying problem is that they lack the abstractions necessary to coordinate CPSs to work at larger scales. Many approaches sought a more holistic view than node-centric programming by introducing abstractions to handle locality via region formation [50, 58], efficient in-network aggregation [29, 40], and shared-memory based on locality [26].

Macroprogramming frameworks were designed to address heterogeneity in the system at scale [6, 7]. They provide abstractions for selecting sets of devices, efficient in-network aggregation, and interfaces between heterogeneous devices in the system. These often take a host-coordination language approach in which the host language, such as C or Java, is used for platform specific code and the coordination language encodes communication channels, message formats, and code location to hardware. These models almost exclusively use message-passing architectures and encode the macroprogram in a graph-based intermediate representation before mapping chunks of code to devices. The devices typically host middleware to handle communication, interfaces, and other common runtime mechanisms that are non-specific to the application. Their functionality reflects many ideas found in Links, which pioneered multitier/tier-

less programming [16]. Links has programmers write client/server applications within a single file and designate functions with annotations where code should be placed. This shortens the distance between functional interfaces from different files (one for a client application and one for the server) into a single file, making it easier to reason about the flow of data in communication between client and server. In this way, Links removed interaction mismatches and centered focus around the programmer's application code. TTPython uses a macroprogramming framework to abstract networking code from the user. The function decorator `@GRAPHify` specifies the interactions between devices, while the context manager `TTConstraint` specifies where functions decorated by `@SQify` and `@STREAMify` go. This forms clear delineations, where TTPython adds timing specifications (periodicity, data synchronization, and time-triggered exception handling code) where they are needed most, between different functional aspects of the code.

Real Time Systems (RTS) frameworks like the Time Triggered Architecture (TTA) [35] and PTIDES [60], which are node-centric and host-coordination approaches, respectively, take a more rigorous approach to timing. These frameworks provide strong guarantees about the behavior of the system and application, a necessity for safety-critical systems, but require deep knowledge of each component of the system. Their approach to time is based heavily on compile-time analysis and formal verification. The devices that compose the system are over-provisioned to ensure that critical components, such as deadlines on motor actuation, are guaranteed to be met. Such over-provisioning becomes more challenging as non-determinism increases, increasing the cost of the system. TTPython focuses more on providing runtime capabilities for the programmer. It provides flexibility for programmers to specify where data is synchronized and when deadlines are checked. TTPython aligns closer with soft real-time systems, where late processing is not completely and immediately catastrophic.

Macroprogramming frameworks have slowly begun to include time as a central component. Early frameworks such as Kairos [26] did not include timing constructs within the framework. They opted to leave timing handled by its host language, leading to intrusive intermingling of application and system behaviors. COSMOS [6] introduced timing at the system-level overview, but lacks time-specification outside of periodicity. Lingua Franca (LF) [39] represents programs as a dataflow graph of actor-like nodes with strong deterministic properties, which helps avoid common error-sources in concurrent systems. LF and PTIDES use "logical time" in each node for deciding when to act upon time-sensitive inputs. Timing uncertainty is estimated for each node at compile time, and inter-node messages may be delayed by the maximum uncertainty to assert in-order-ness of inputs. They introduce timing constructs to specify periodicity and deadlines at the function level or their relation across direct edges between actors. This makes it difficult to understand global timing specifications. The programmer needs to intuit global timing specifications by understanding actor code and how time changes across edges. TTPython prevents these timing specifications from being split by managing timing and deadlines as SQ firing rules and explicit nodes for time management. The programmer has the option to implement their own timing information not managed by TTPython within a SQ.

Synchronous languages like Esterel [8], Lustre [27], and SIGNAL [22] provide programmer-friendly ways to reason about time and synchrony in real-time, parallel systems by using signal-based abstractions. These languages use logical clocks to hide explicit time management of timestamps. This turns inputs and outputs into time-triggered signals, which are an effective model for many cyber-physical applications. Synchrony is achieved by designating a generic input signal as a clock and starting computation based on the availability of the signal. These languages also commonly follow a data-flow paradigm, which TTPython employs as well. In contrast, Functional Reactive Programming offers a principled approach in extending a host language with notions of time flow by using types, as seen in Fran [23] and Yampa [18]. It uses both continuous and logical time to support different types of inputs depending on the input's

tendency to change values. Work has been done to integrate these timing-focused languages with tierless programming. HipHop takes synchronous reactive programming and showcases how these translate well in a web programming environment. Reynders et al. proposed combining multi-tier ideas with tierless programming for client-server architectures. ScalaLoci [57] expands on this with placement types to describe the data communicated across different actors, allowing it to express a wider range of applications in a tierless model. The ideas of event abstractions appear in TTPython with its time-triggered exception handling code. TTPython's deadline model relaxes the notion of simultaneity as synchronization between its control and data planes. When applying these ideas with other timing specifications, however, these languages and paradigms do not translate well. Global timing dependencies must be split over multiple devices to realize, which makes it difficult to reconstruct the intended specification. These languages struggle in interfacing between heterogeneous compositions of distributed devices in the network.

Dataflow graphs were first introduced in the seminal paper by Dennis [21]. Its key insight was identifying that shared data bottlenecks parallelism; therefore, parallel computation should be guided by the availability of data. The semantics were later refined with the U-Interpreter paper [46] by assigning labels to data, allowing nodes to work asynchronously from each other. These labels allowed for-loop execution, as the labels identify which data corresponds to a specific iteration. Data must share the label's context before it can be used for execution. These ideas were later refined in the MIT Token-Tagged Dataflow paper [51] to work well within the context of a von Neumann machine. Our work builds on the foundation of token-tagged dataflow and introduces time as a specific "tag," which allows us to extend dataflow to account for DT applications.

Timing in distributed systems has been represented in various ways. One such representation is to view timestamps not as singular values, but as distributions, of which uniform intervals are the most common [5, 17, 41]. In the TrueTime API within Google Spanner [17], time intervals are used to establish global event ordering and versioning for broadly distributed and replicated databases. This is similar to the purpose of Lamport and Vector Clocks [36, 38], but requires fewer message exchanges between servers, instead relying on time synchronization services like GPS, IEEE 1588 (PTP) [2], or NTP [45]. These intervals represent an uncertainty of when the computation happened. This idea aligns with TTPython's execution rules using time intervals. In TTPython, data is assigned a time interval. This gives it a context in which it can be used with other data. An interpretation is that data was generated sometime within this time interval, so it can be treated as synchronously generated with other data that overlap with this time interval. TTPython's abstractions with pairing data with time intervals can mitigate the effects of time jitter and gradual desync, but work best when paired with a time synchronization service.

Dataflow graphs have also incorporated other types of timing abstractions. Naiad [48] includes a logical timestamp within its records (tokens) to order tokens in a distributed fashion. This prevents it from performing out-of-order execution. The timed, tagged-token dataflow graph opts to use physical timestamps to increase asynchrony across its devices. A logical timeline requires agreed synchronization on generated data to make it unambiguous which data can be used with it. Physical timestamps retain more timing information and allow separation between the data generation and its sinks. This flexibility makes multitenancy much easier to implement, as data streams can be shared across different applications without having to synchronize outputs for each specific application. This is a future goal for TTPython.

5.2 Human-centered Programming Language User Studies

Programming languages have begun to look at Human-Computer Interaction techniques to help evaluate language design and usability. Coblenz et al. [15] provides a process and framework for iterative, human-centered language design. This technique has given rise to two new languages, Glacier [13] and

Obsidian [14]. These have also inspired usability studies on identifying challenges associated with integrating new language paradigms into popular languages, such as Liquid Types in Haskell [24]. However, programmer user studies are still very challenging due to large investments in time to develop domain knowledge of study design and to create one [19].

User studies have also been extended to examine the usability of concurrency and distribution in language design. Nanz et al. [49] proposed an experimental design between two object-oriented languages designed for concurrency (Java and SCOOP). This was conducted in a Software Architecture class setting in a 4-hour controlled setting and laid the foundation for larger-scale studies. Hochstein et al. [30] took a different approach by comparing different parallel programming models (message-passing vs. parallel random-access machine) through a take-home assignment. A holistic study [3] compared 69 participants using distributed data processing platforms for different data science problems over the course of the semester. They collected preference ranking and System Usability Survey [11] scores for each participant to quantify usability results. To our knowledge, there is no prior user study on testing timing specifications.

Chapter 6

Conclusion

Without the domain knowledge of CPSs, many programmers would struggle to use the advantages of in-field sensor networks and embedded systems. One of the major benefits of DT applications includes the assistance in data gathering, reacting, and understanding for in-field experts of natural phenomena, such as meteorology and traffic control. Sensor networks vastly improve the quantity of data while keeping quality at a reasonable level. Distributed systems interacting with the physical world require a timeline not only for ordering events but also for coordinating physical I/O and determining concurrency to compare or combine data values. The technical knowledge to program devices in a heterogeneous network while accounting for timing specifications is challenging.

Designing applications for cyber-physical systems is difficult due to various distribution and time-sensitive requirements. These issues stem from working with heterogeneous devices that asynchronously cooperate in real-time. We identified common distribution and timing patterns found in these DT applications through two current university research projects. The smart intersection application provided an opportunity to compare TTPython's capabilities with a vanilla Python implementation. The urban flooding application provided use cases to compose TTPython's constructs and to explore its expressiveness in wireless sensor networks. These patterns led to the creation of TTPython, a language and system designed to abstract these patterns found in DT applications. Our system reduces barriers to entry in understanding how to coordinate communication and timing between multiple devices while providing safety mechanisms for the programmer. It provides timing constructs to create asynchronous data streams, synchronize and mutually exclude data, and provide time-triggered exception handling. Its macroprogramming design abstracts networking decisions to variable assignments and makes it simple to move code across devices. Under the hood, TTPython uses a timed, tagged-token dataflow graph architecture to realize these abstractions. We provide a compilation and semantics for this novel execution style. TTPython's abstractions were tested with a within-subjects user study to determine their effect. The task design for the user study was inspired by TTPython's prior case studies. We found that these abstractions in isolation do well in comparison to their vanilla Python counterpart.

Appendix A

Appendix

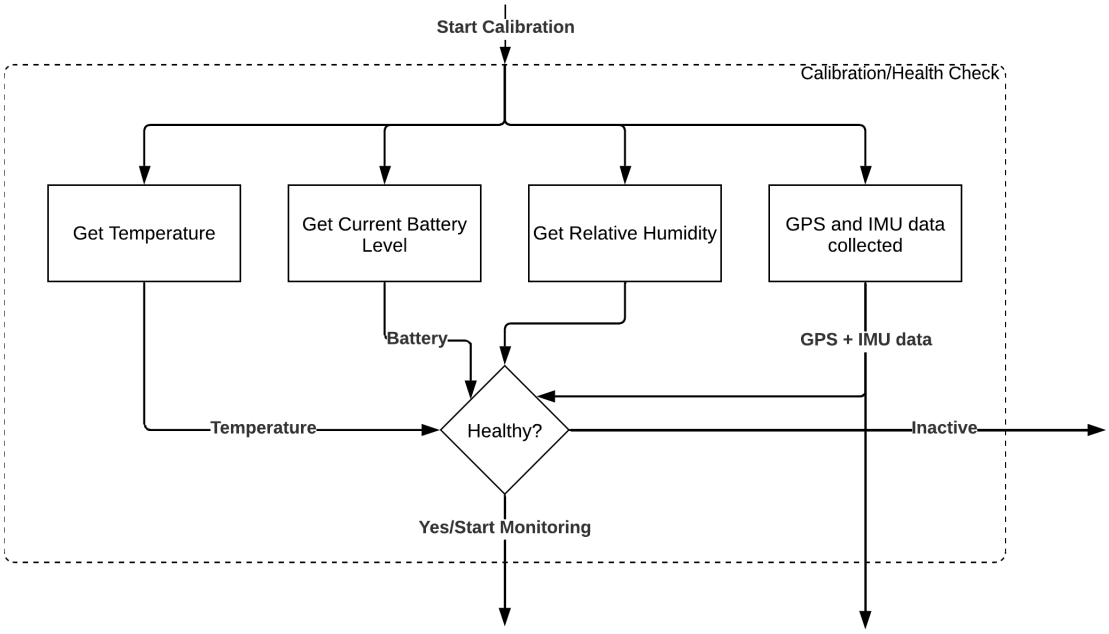


Figure A.1: Calibration mode in the state transition graph.

A.1 User Study Materials

The following materials show what the participant in the qualitative user study sees as starter code. Some of the code that the user does not interact with has been omitted.

A.1.1 SI App Introduction

The goal of this application is to create a smart intersection where connected autonomous vehicles (CAV) (pictured above) communicate with each other to navigate through a lightless traffic intersection. A device named the roadside unit (RSU) assists the vehicles by fusing their data before choosing one of them to

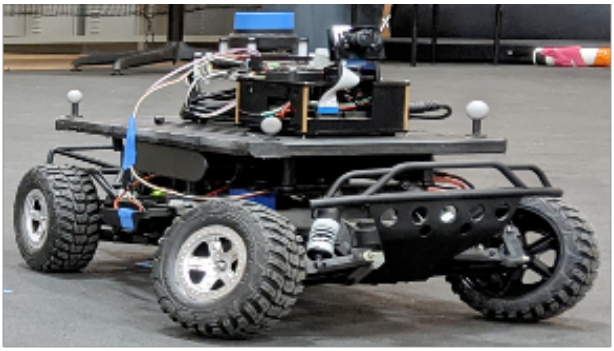


Figure A.2: A connected autonomous vehicle (CAV).

move through the intersection. The CAVs first take an image and LIDAR sample, and fuse them together with `local_fusion`. They send it to the RSU, which will globally fuse data from all CAVs. The RSU then returns movement instructions on what to do back to the CAVs that will then actuate their motors in response.

A.1.2 UF App Introduction

The goal of this application is to create code for urban deployed devices to detect flooding. There are three devices: an optical camera device, a thermal camera device, and a router. The created optical image is tagged with geolocation information from the IMU and GPS sensors. The thermal and optical camera image is then coregistered (superimposed) by the thermal camera device. The combined image is then sent over the network to a programmable router that hosts a machine learning model to detect if flooding occurs in the image. The result is then saved on a remote server.

Figure A.3: An optical camera device.

A.1.3 Task ADG

SI App

In this task, you will write code to set up a camera data stream, one of the three sensors on the optical camera device. You will create a camera process that will call `take_image`, which periodically generates images and sends them back asynchronously to the main process through an interprocess Queue.

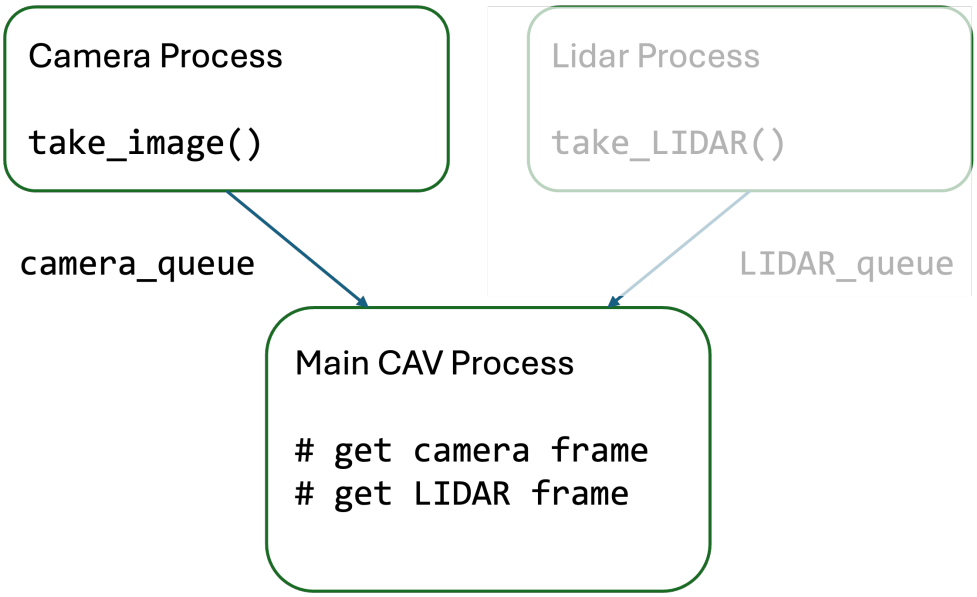


Figure A.4: Vanilla Python ADG architecture design of the CAV.

You will do the following:

- Approximate Data Timestamps
- Make a periodic loop.
- Send images through an interprocess Queue.

In the next Jupyter Notebook for Task 1:

- Create the camera process in the main process and read from the optical camera queue.

```
###### P Task 1: Part 1
# Create an approximated timestamp of when `func`s output was produced (say
# taking an image). We time the start and end of `func`, average it, and add
# a tolerance to account for hardware jitter. A visual is shown below:
#               A                                   B
#          time.time()         func()          time.time()
#              |                 |                 |
# time ----------------------------------------------------------->
#       |                        |                         |
#        <---------------------> <----------------------->
```

```
#                tolerance                  tolerance
# The interval's midpoint is defined (A + B)/2. `time_interval` is a list of
# length 2 with [`start`, `end`].
# `start` is defined as midpoint - tolerance.
# `end` is defined as midpoint + tolerance
def approximate_data_timestamp(func, tolerance):
    # TODO: fill in the `time_interval` below.
    data = func()
    time_interval = [..., ...]
    print((data, [time_to_str(t) for t in time_interval]))
    return (data, time_interval)

# period is in seconds
def take_image(camera_queue: mp.Queue, start_time, period, tolerance):
    ...

    # inits camera
    camera = DataSources.Camera()

    ###### P Task 1: Part 2
    # Make a periodic call to `approximate_data_timestamp` with a period of
    # parameter `period` (unit: seconds) and continue indefinitely. Execute
    # every `period` seconds relative from the start (i.e. period=1, runs at
    # 1:00, 1:01, 1:02). Do not worry about the case when the code takes
    # longer than the period to complete.

    # TODO: Make this code periodic
    timestamped_data = approximate_data_timestamp(camera.take_camera_frame,
                                                  tolerance)
    ###### P Task 1: Part 3
    # TODO: in your created periodic loop above, put `timestamped_data` into
    # the interprocess queue `camera_queue`.

    ###### P Task 1: Part 3

    ###### P Task 1: Part 2

    # NOTE: you may use this to test your queue in your periodic loop.
    # print(f'received: {camera_queue.get()[0]}')
```

Listing 24: Part 1 of the starting code for the Python ADG Task of the SI app.

In this Jupyter Notebook for Task 1:

- Create the camera process in the main process and read from the optical camera queue.

```
def cav_main():

    ###### P Task 1: Part 4
    # TODO: Create an interprocess Queue.

```

```
    # TODO: Create and start the camera process.
    # target is `take_image`, args are the queue you made, `start_time` as
    # defined below, period of 1 second, and tolerance of 0.250 seconds
    start_time = time.time() + 1
    period = 1.0
    tolerance = 0.25

    ###### P Task 1: Part 4

    ...

    try:
        ###### P Task 1: Part 5
        # TODO: Make a periodic loop of period 1 secnod. Execute every 1
        # second relative from the start (i.e. runs at 1:00, 1:01, 1:02,
        # ...).

        # TODO: In the periodic body loop, try to get from the camera
        # process's queue until either 0.5 seconds (fallthrough_delay) passes
        # or you get a value from the queue and print it. Make sure that
        # reading the queue is a non blocking operation.
        fallthrough_delay = 0.5

        # TODO: if you get a value from the queue, print it
        # print('received: {value}') # uncomment me!

        ###### P Task 1: Part 5
```

Listing 25: Part 2 of the starting code for the Python ADG Task of the SI app.

In this task, you will write code to set up a camera data stream for a CAV. You will make `take_image` generate images periodically and asynchronously.

You will do the following:

- Make `take_image` periodic and call it in `@GRAPHify`.

```
def take_image(trigger):
    from libs import DataSources
    global sq_state

    # Init the camera class
    if sq_state.get('cam_recog', None) is None:
        sq_state['cam_recog'] = DataSources.Camera()
    cameraRecognition:DataSources.Camera = sq_state['cam_recog']

    value = cameraRecognition.take_camera_frame()
    print(f'got: {value}')

    return value

```

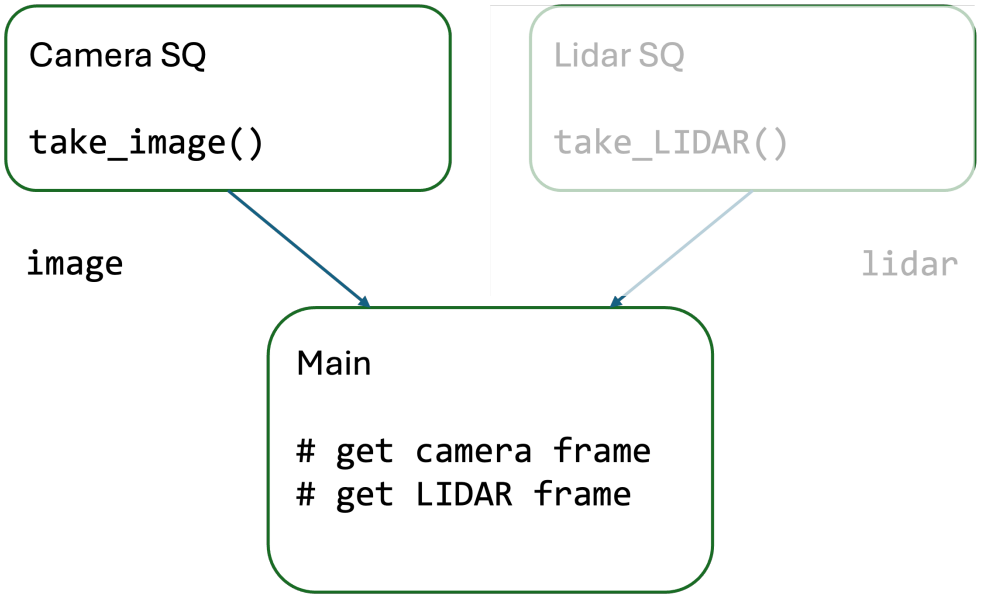


Figure A.5: TTPython ADG architecture design for the CAV.

```
@GRAPHify
def cav(trigger):
    with TTClock.root() as root_clock:
        start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 2_000_000
        sampling_time = VALUES_TO_TTTIME(start_time,
                                         GET_INFINITY(trigger,
                                                      TTClock=root_clock))
        sample_window = COPY_TTTIME(trigger, sampling_time)

        ###### T Task 1:
        # TODO: Make `take_image` periodic and call it with a period of 1
        # second. Have `take_image` use `sample_window` as its trigger.
        # Use a TTDataIntervalWidth of 500_000 microseconds and set
        # TTPhase=0.
```

Listing 26: The starting code for the TTPython ADG Task of the SI app.

UF App

In this task, you will write code to set up a camera data stream, one of the three sensors on the optical camera device. You will create a camera process that will call `take_opt_image`, which periodically generates images and sends them back asynchronously to the main process through an interprocess Queue.

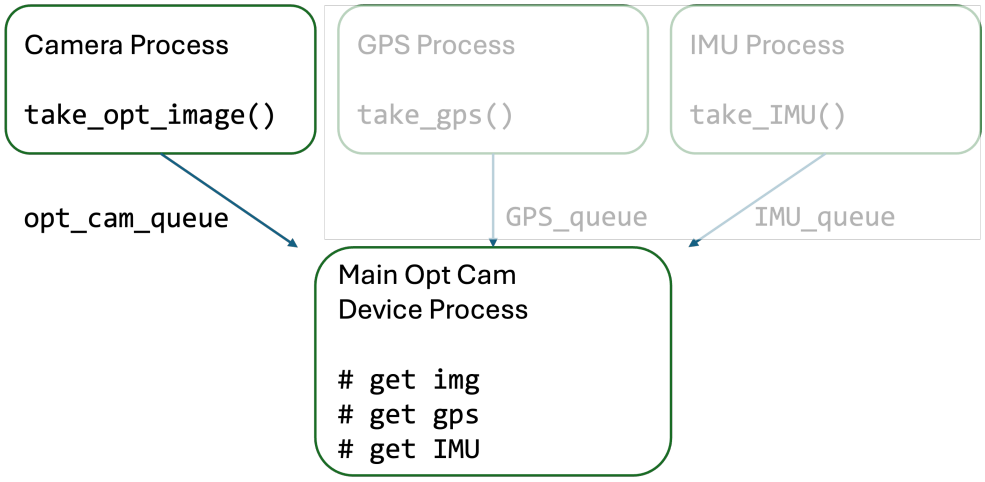


Figure A.6: Vanilla Python ADG architecture design for the optical camera.

You will do the following:

- Approximate Data Timestamps
- Make a periodic loop.
- Send images through an interprocess Queue.

In the next Jupyter Notebook for Task 1:

- Create the camera process in the main process and read from the optical camera queue.

```
###### P Task 1: Part 1
# Create an approximated timestamp of when `func`s output was produced (say
# taking an image). We time the start and end of `func`, average it, and add
# a tolerance to account for hardware jitter. A visual is shown below:
#              A                                  B
#          time.time()        func()         time.time()
#              |                |                |
# time ----------------------------------------------------->
#       |                       |                       |
#        <---------------------> <--------------------->
#               tolerance               tolerance
# The interval's midpoint is defined (A + B)/2. `time_interval` is a list of
# length 2 with [`start`, `end`].
# `start` is defined as midpoint - tolerance.
# `end` is defined as midpoint + tolerance
```

```python
def approximate_data_timestamp(func, tolerance):
    # TODO: fill in the `time_interval` below.
    data = func()
    time_interval = [..., ...]
    print((data, [time_to_str(t) for t in time_interval]))
    return (data, time_interval)

# period is in seconds
def take_opt_image(opt_cam_queue: mp.Queue, start_time, period, tolerance):
    # waits until start_time
    ...

    # inits opt_cam
    opt_cam: DataSources.Camera = DataSources.Camera()

    # Make a periodic call to `approximate_data_timestamp` with a period of
    # parameter `period` (unit: seconds) and continue indefinitely. Execute
    # every `period` seconds relative from the start (i.e. period=2, runs at
    # 1:00, 1:02, 1:04). Do not worry about the case when the code takes
    # longer than the period to complete.

    # TODO: Make this code periodic
    timestamped_data = approximate_data_timestamp(opt_cam.take_camera_frame,
                                                  tolerance)
    # TODO: in your created periodic loop, put `timestamped_data` into the
    # interprocess queue `opt_cam_queue`.
```

Listing 27: Part 1 of the starting code for the vanilla Python ADG Task of the UF app.

```python
def opt_main():
    ###### P Task 1: Part 4
    # TODO: Create an interprocess Queue.

    # TODO: Create and start the camera process.
    # target is `take_opt_image`, args are the queue you made,
    # `start_time` as defined below, period of 2 seconds,
    # and tolerance of 0.250 seconds
    start_time = time.time() + 1
    tolerance = 0.25
    period = 2

    ###### P Task 1: Part 4

    ### Wait until the start_time to start executing
    wait_until_start = start_time - time.time()
    if 0 < wait_until_start:
        time.sleep(wait_until_start)
    ###

    ###### P Task 1: Part 5
    # TODO: Make a periodic loop of two seconds. Execute every 2 seconds
```

```
    # relative from the start (i.e. runs at 1:00, 1:02, 1:04).

    # TODO: In the periodic body loop, try to get from the camera process's
    # until either 1 second (fallthrough_delay) passes or you get
    # a value from the queue and print it. Make the reading from
    # the queue is a nonblocking operation.
    fallthrough_delay = 1

    # TODO: if you read a value from the queue, print it
    # print(f'received: {value}') # uncomment me!

    ###### P Task 1: Part 5
```

Listing 28: Part 2 of the starting code for the vanilla Python ADG Task of the UF app.

In this task, you will write code to set up a camera data stream for a CAV. You will make `take_opt_image` generate images periodically and asynchronously.

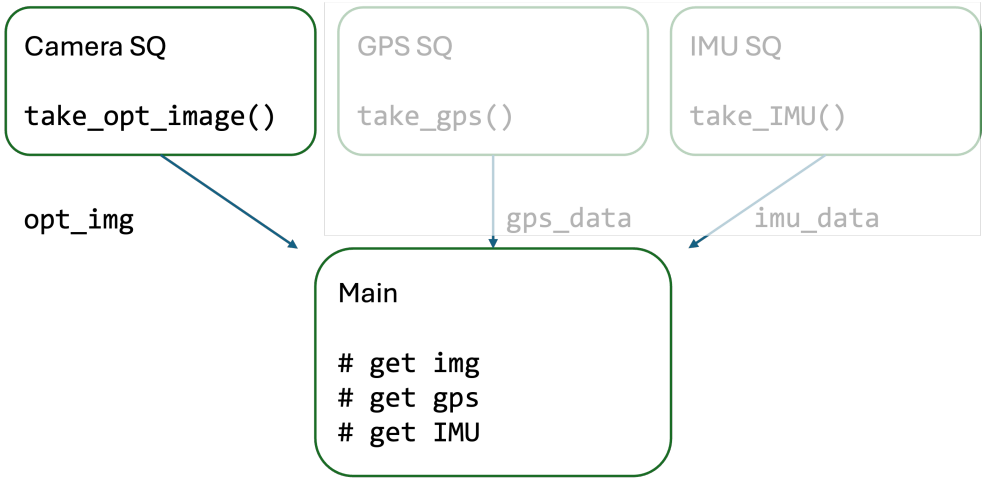


Figure A.7: ADG architecture design for the optical camera device.

You will do the following:

- Make `take_opt_image` periodic and call it in `@GRAPHify`.

```
def take_opt_image(trigger):
    from libs import DataSources
    global sq_state

    # Init the optical camera
    if sq_state.get('opt_cam', None) is None:
        sq_state['opt_cam'] = DataSources.Camera()
    opt_cam: DataSources.Camera = sq_state['opt_cam']
    value = opt_cam.take_camera_frame()
    print(f'got: {value}')
```

```
11
12      return value
13
14  @GRAPHify
15  def main(trigger):
16      with TTClock.root() as root_clock:
17          start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 1_000_000
18          sampling_time = VALUES_TO_TTTIME(
19              start_time, GET_INFINITY(trigger, TTClock=root_clock))
20          sample_window = COPY_TTTIME(trigger, sampling_time)
21
22          ###### T Task 1:
23          # TODO: Make `take_opt_image` periodic and call it with a period of 2
24          # seconds. Have `take_opt_image` use `sample_window` as its trigger.
25          # Use a TTDataIntervalWidth of 500_000 microseconds and set TTPhase=0.
```

Listing 29: The starting code for the TTPython ADG Task of the UF app.

A.1.4 Task DS

```python
class Interval:
    # you can assume that start <= end
    def __init__(self, start, end):
        self.start = start
        self.end = end

        if end < start:
            raise Exception("Invalid interval range")

    def __str__(self):
        return (f'Interval:[{DataGen.time_to_str(self.start)}, '
                f'{DataGen.time_to_str(self.end)}]')

    def to_list(self):
        return [self.start, self.end]

class TimedData:

    def __init__(self, data, interval: List[float]):
        self.data = data
        self.interval = Interval(interval[0], interval[1])

    def __repr__(self):
        return f'TimedData<data:{self.data}, {self.interval}>'

if __name__ == "__main__":
    x_interval = Interval(0, 10)
    y_interval = Interval(5, 15)
    z_interval = Interval(7, 17)
    outside_interval = Interval(30, 50)

    x = TimedData(0, x_interval.to_list())
    # access its Interval through `x.interval`
    y = TimedData(1, y_interval.to_list())
    z = TimedData(2, z_interval.to_list())
    outside = TimedData(4, outside_interval.to_list())

    set_1: set[TimedData] = set()
    set_2: set[TimedData] = set()
    set_3: set[TimedData] = set()
    set_outside: set[TimedData] = set()

    set_1.add(x)
    set_2.add(y)
    set_3.add(z)
    set_outside.add(outside)
```

Listing 30: Shared time library and testing code for the Python DS Task of the SI app.

SI App

The CAV generates image and LIDAR data asynchronously. We want to first synchronize data before using it, as an image from the camera and data from the LIDAR if used together should be taken at about the same time. We say they are **synchronized** when the time intervals associated with the two pieces of data overlap.

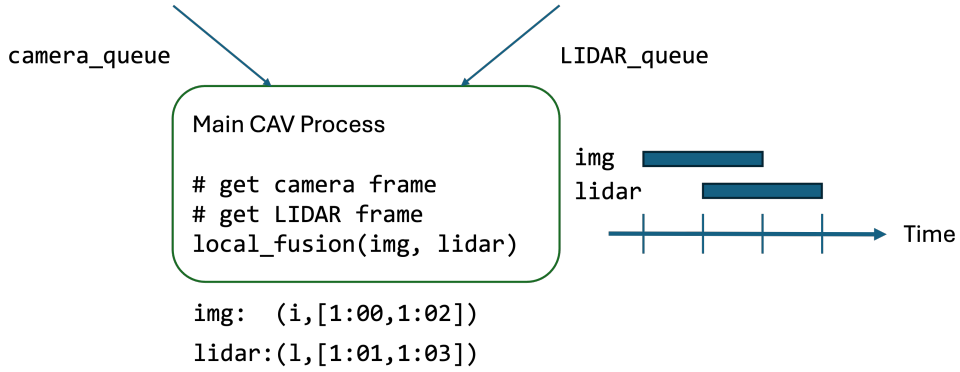


Figure A.8: Data Synchronization between camera and LIDAR.

You will do the following.

- Determine if two intervals overlap in time. You can assume that endpoints are inclusive.
- Find synchronized data between 2 sets.
- Call `local_fusion` with a synchronized image and LIDAR data.

```
###### P Task 2: Part 1
# Check whether two intervals are overlapping in time (i.e. synchronized)
def overlapping_timestamps(interval_x: Interval,
                           interval_y: Interval) -> bool:
    # TODO: return True if there is an overlap of time intervals
    return False

###### P Task 2: Part 2
# Check from 2 sets if there is synchronized data (data that overlaps in
# time with each other), and return that pair if so. return any arbitrary
# pair.
def get_overlapping_data(
        data_set_1: set[TimedData],
        data_set_2: set[TimedData]) -> Optional[tuple[TimedData, TimedData]]:
    # TODO: returns (data_1, data_2) where `data_1` and `data_2`'s time
    # interval overlap and `data_{num}` is in `data_set_{num}`, otherwise
    # return None
    return None

###### P Task 2 : Part 3
# This function removes synchronized data from its respective set if there
```

```
# is a synchronized set of data
def extract_overlapping_data(
        data_set_1: set[TimedData],
        data_set_2: set[TimedData]) -> Optional[tuple[TimedData, TimedData]]:
    # TODO: call `get_overlapping_data`.
    # TODO: If the return is not None, it will return (data_1, data_2).
    # Remove `data_{num}` from each `data_set_{num}` respectively.
    # TODO: Return output from `get_overlapping_data`.
    return None
```

Listing 31: Part 1 of the starting code for the Python DS Task of the SI app.

In the body of `cav_main`, you will extract overlapping data from the camera and LIDAR. If there is a synchronized match, you will assign them to their corresponding variables. The output should be a tuple of `(data, time_interval)` where `data` is a list of 2 elements each having the same value increasing in time. A sample output is given below.

```
def cav_main():
    ...

    images = set()
    lidars = set()
    try:
        # periodic loop
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                ###### P Task 2: Part 4
                # TODO: Extract an overlapping pair of image and lidar data
                # from `images` and `lidars`. If a set of data is synchronized
                # (return is not None), assign it to `image` and `lidar`
                # respectively (otherwise set them to None). They will be used
                # in the call to `local_fusion(...)`.
                image = None
                lidar = None

                ###### P Task 2: Part 4

                if (image is not None and lidar is not None):
                    image: TimedData
                    lidar: TimedData
                    lf0 = local_fusion(image.data, lidar.data)
                    interval = intersect_list(image.interval.to_list(),
                                              lidar.interval.to_list())
                    print(
                        f'local fusion: {(lf0, [time_to_str(t) for t in interval])}'
                    )
                else:
                    print(f'local fusion: No synchronized data')
```

```
35
36                          ...
```

Listing 32: Part 2 of the starting code for the Python DS Task of the SI app.

The CAV generates image and LIDAR data asynchronously. We want to first synchronize data before using it, as an image from the camera and data from the LIDAR if used together should be taken at about the same time. We say they are **synchronized** when the time intervals associated with the two pieces of data overlap.

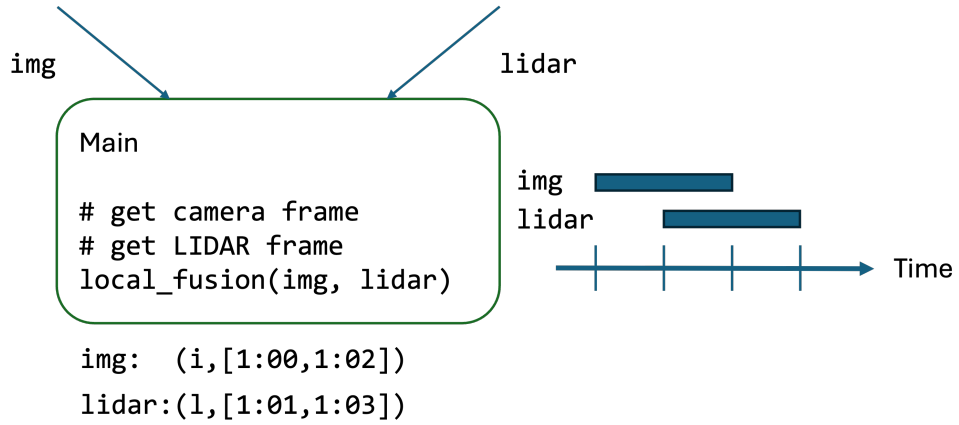


Figure A.9: Data Synchronization between camera and LIDAR.

You will do the following:

- Call `local_fusion` in the GRAPHified function `cav`.
- Use TTPython's synchronization rules to make sure data it operates on is synchronized in time.

```python
@STREAMify
def take_image(trigger):
    ...

@STREAMify
def take_LIDAR(trigger):
    ...

def local_fusion(image, lidar):
    if image != lidar:
        print(f'cav0 img: {image} not same as lidar: {lidar}')
    else:
        print(f'{[image, lidar]}')

    return [image, lidar]

@GRAPHify
```

```
def local_fusion_ttpy(trigger):
    with TTClock.root() as root_clock:
        start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 1_000_000
        sampling_time = VALUES_TO_TTTIME(
            start_time, GET_INFINITY(trigger, TTClock=root_clock))
        sample_window = COPY_TTTIME(trigger, sampling_time)

        image = take_image(sample_window,
                           TTClock=root_clock,
                           TTPeriod=1_000_000,
                           TTPhase=0,
                           TTDataIntervalWidth=250_000)
        lidar = take_LIDAR(sample_window,
                           TTClock=root_clock,
                           TTPeriod=1_000_000,
                           TTPhase=0,
                           TTDataIntervalWidth=250_000)

        ###### Task 2:
        # TODO: Call `local_fusion` with `image` and `lidar`. Ensure that
        # these values are synchronized (their data intervals
        # overlap in time).
```

Listing 33: The starting code for the TTPython DS Task of the SI app.

UF App

The optical camera has processes asynchronously generating GPS and IMU data to provide metadata to the optical image (its EXIF data). We want to first synchronize data before using it, as an image from the camera and data from the GPS and IMU if used together should be taken at about the same time. We say they are **synchronized** when the time intervals associated with the two pieces of data overlap.

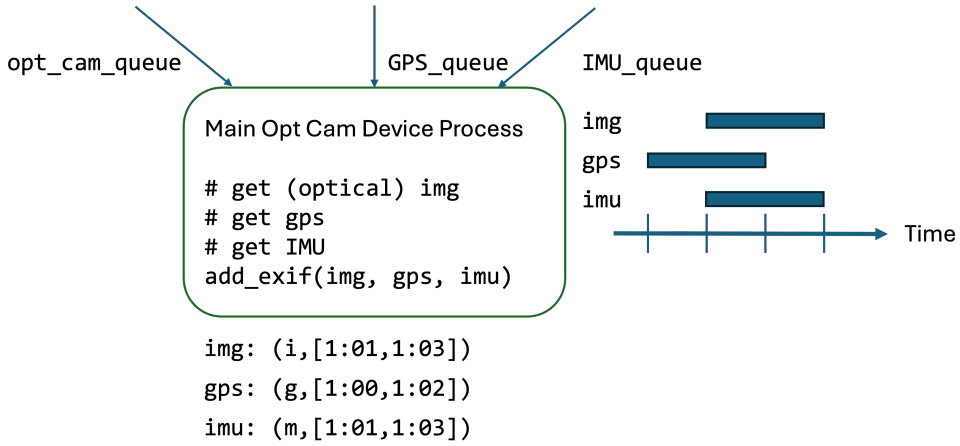


Figure A.10: Data Synchronization between optical image, GPS, and IMU.

You will do the following.

- Determine if two intervals overlap in time. You can assume that endpoints are inclusive.
- Find synchronized data between 3 sets.
- Call `add_exif` with a synchronized optical image, GPS, and IMU data.

```python
###### P Task 2: Part 1
# Check whether two intervals are overlapping in time (i.e. synchronized)
def overlapping_timestamps(interval_x: Interval,
                           interval_y: Interval) -> bool:
    # TODO: return True if there is an overlap of time intervals
    return False

###### P Task 2: Part 2
# Check from 3 sets if there is synchronized data (data that overlaps in time
# with each other), and return that trio if so. return any arbitrary trio.
def get_overlapping_data(
    data_set_1: set[TimedData], data_set_2: set[TimedData],
    data_set_3: set[TimedData]
) -> Optional[tuple[TimedData, TimedData, TimedData]]:
    # TODO: returns (data_1, data_2, data_3) where data_1, data_2, and
    # data_3's time interval overlap and data_{num} is in data_set_{num},
    # otherwise return None
    return None

```

```python
###### P Task 2 : Part 3
# This function removes synchronized data from its respective set if there
# is a synchronized set of data
def extract_overlapping_data(
    data_set_1: set[TimedData], data_set_2: set[TimedData],
    data_set_3: set[TimedData]
) -> Optional[tuple[TimedData, TimedData, TimedData]]:
    # TODO: call `get_overlapping_data`.
    # TODO: If its return is not None, it will return (data_1, data_2,
    # data_3). Remove data_{num} from each data_set_{num} respectively.
    # TODO: Return output from `get_overlapping_data`.
    return None
```

Listing 34: Part 1 of the starting code for the vanilla Python DS Task of the UF app.

In the body of `opt_main`, you will extract overlapping data from the optical camera, gps, and imu sets. If there is a synchronized match, you will assign them to their corresponding variables. The output should be a tuple of `(data, time_interval)` where `data` is a list of 3 elements each having the same value increasing in time. A sample output is given below.

```python
def opt_main():
    ...

    opt_imgs = set()
    imus = set()
    gpss = set()

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                fallthrough = curr_time + fallthrough_delay

                ###### P Task 2: Part 4
                # TODO: Extract an overlapping trio of camera, imu, and
                # gps data from `opt_imgs`, `imus`, and `gpss` respectively.
                # If a set of data is synchronized (return is not None),
                # assign it to `opt_img`, `imu_data`, and `gps_data`
                # respectively. They will be used in the call to
                # `add_exif(...)`
                opt_img = None
                imu_data = None
                gps_data = None

                ###### P Task 2: Part 4

                if (opt_img is not None and imu_data is not None
                        and gps_data is not None):
                    opt_img: TimedData
                    imu_data: TimedData
                    gps_data: TimedData
```

```
                    exif_img = add_exif(opt_img.data, imu_data.data,
                                        gps_data.data)
                    interval = intersect_list(
                        intersect_list(opt_img.interval.to_list(),
                                       imu_data.interval.to_list()),
                        gps_data.interval.to_list())
                    print(
                        f'exif: {(exif_img,
                                 [time_to_str(t) for t in interval])}'
                    )
                else:
                    print(f'exif: No synchronized data')

                ...
```

Listing 35: Part 2 of the starting code for the vanilla Python DS Task of the UF app.

The optical camera has processes asynchronously generating GPS and IMU data to provide metadata to the optical image (its EXIF data). We want to first synchronize data before using it, as an image from the camera and data from the GPS and IMU if used together should be taken at about the same time. We say they are **synchronized** when the time intervals associated with the two pieces of data overlap.

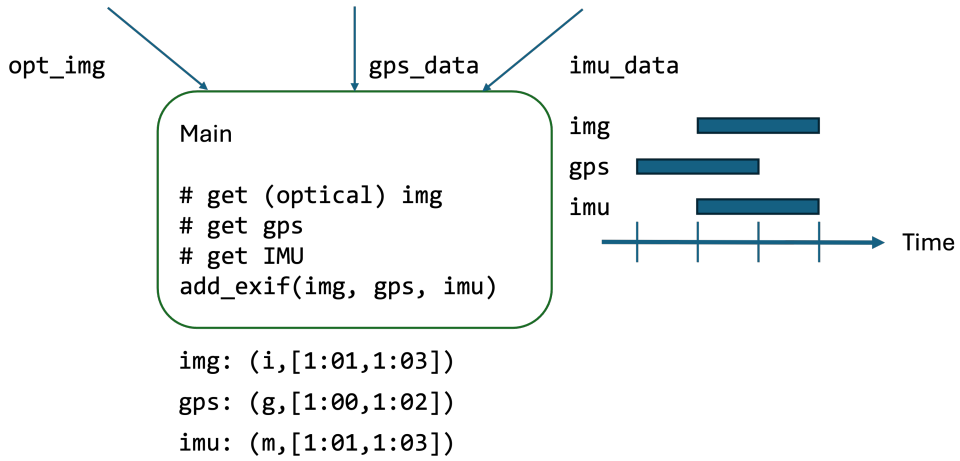


Figure A.11: Data Synchronization between optical image, GPS, and IMU.

You will do the following:

- Call `add_exif` in the GRAPHified function `main`.
- Use TTPython's synchronization rules to make sure data it operates on is synchronized in time.

```
...

def add_exif(opt_img, gps_exif, imu_data):
    if opt_img != gps_exif or opt_img != imu_data:
```

```
        print(f"data doesn't match -> opt_img: {opt_img}, "
              f'gps: {gps_exif}, imu: {imu_data}')
    else:
        print(f'add_exif: {[opt_img, gps_exif, imu_data]}')

    return [opt_img, gps_exif, imu_data]

@GRAPHify
def main(trigger):
    with TTClock.root() as root_clock:
        start_time = READ_TTCLOCK(trigger, TTClock=root_clock) + 1_000_000
        sampling_time = VALUES_TO_TTTIME(start_time,
                                         GET_INFINITY(trigger, TTClock=root_clock))
        sample_window = COPY_TTTIME(trigger, sampling_time)

        opt_image = take_opt_image(sample_window,
                                   TTClock=root_clock,
                                   TTPeriod=2_000_000,
                                   TTPhase=0,
                                   TTDataIntervalWidth=500_000)
        gps_data = take_gps(sample_window,
                            TTClock=root_clock,
                            TTPeriod=2_000_000,
                            TTPhase=0,
                            TTDataIntervalWidth=500_000)
        imu_data = take_imu(sample_window,
                            TTClock=root_clock,
                            TTPeriod=2_000_000,
                            TTPhase=0,
                            TTDataIntervalWidth=500_000)

        ###### T Task 2
        # TODO: Call `add_exif` with `opt_image`, `gps_data`, and `imu_data`.
        # Ensure that these values are synchronized, or that their data
        # intervals overlap in time.
```

Listing 36: The starting code for the TTPython DS Task of the UF app.

A.1.5 Task NTWK

SI App

You will implement the network capabilities between the CAV and the RSU. The CAV wants to send the output of `local_fusion` to the RSU for its call to `global_fusion`. The RSU then figures out each CAVs next direciton with `calculate_angle` and sends it back to them. The CAV then calls `command_motors` with that information.

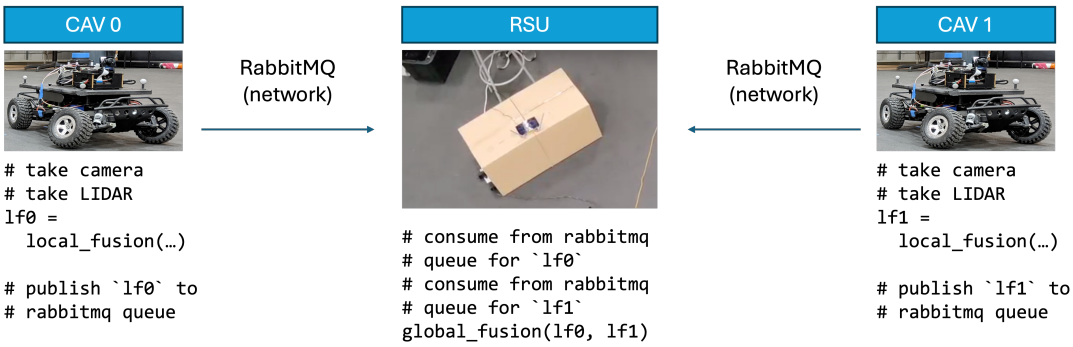


Figure A.12: System architecture describing CAV to RSU communication.

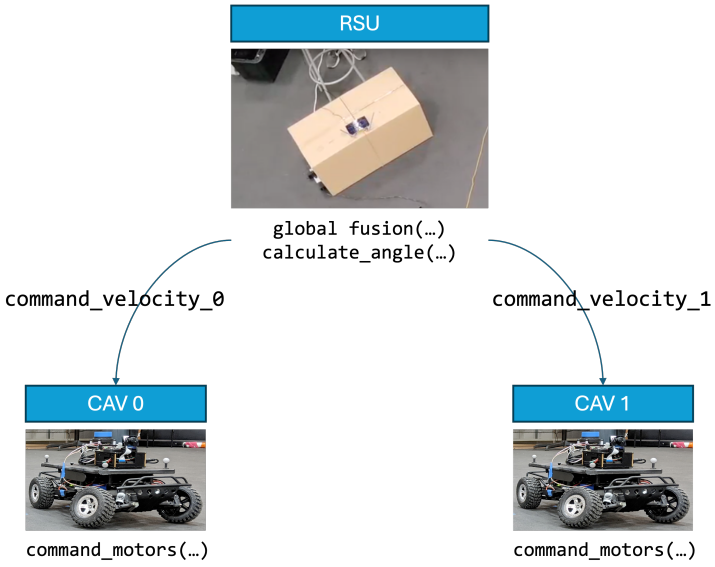


Figure A.13: System architecture describing RSU to CAV communication.

You will be using RabbitMQ as a message broker to communicate between devices. Each device will create processes to listen for incoming data on the **network** queue. This is because RabbitMQ connections are blocking, and each device needs to run periodically. The process will take the data from the network and send it to the main process through an **interprocess** queue.

You will do the following:

- Initialize RabbitMQ connection on `"cav0"`, publish `lf0` to `"rsu"`, receive from `"rsu"` on RabbitMQ connection, and set up processes and queues to read from RabbitMQ.

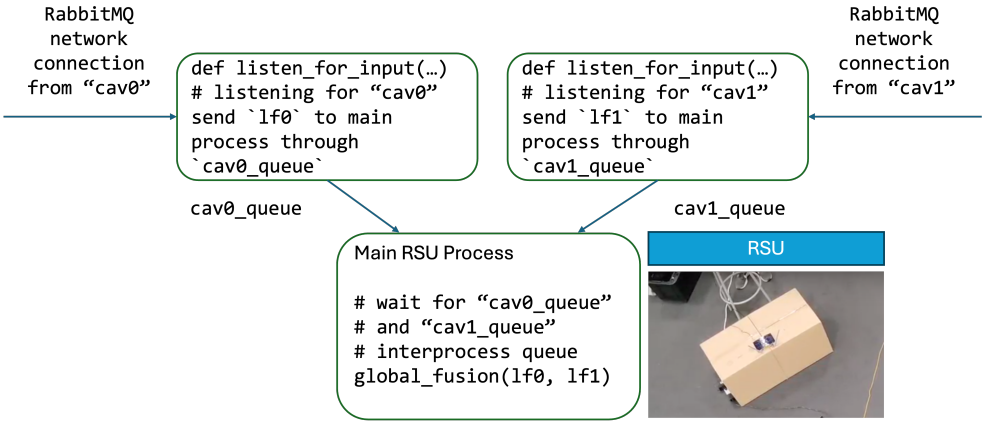


Figure A.14: System architecture describing RabbitMQ interface.

- Initialize RabbitMQ connection on `"cav1"`, publish `lf1` to `"router"`, receive from `"rsu"` on RabbitMQ connection, and set up processes and queues to read from RabbitMQ.
- Initialize RabbitMQ connection on `"router"`, receive from `"cav0"` and `"cav1"` on RabbitMQ connection, and set up processes and queues to read from RabbitMQ.

You will modify `task3_p_cav0.py`, `task3_p_cav1.py`, and `task3_p_rsu.py`. It is suggested you modify the files in this order. You can use the cell below to test your code.

```
def cav0_main(start_time):
    ...

    ###### P Task 3
    # TODO: initialize a RabbitMQ queue connection to device "rsu"
    # As a reminder...
    # * instantiate a new pika blocking connection connected to localhost.
    # * Get the channel from the connection.
    # * Use the default exchange (''), declare the queue with a name of your
    # choice.
    ###### P Task 3: Part 1.

    # NOTE: Skip the initialization task below until you finish the
    # implementation for `listen_for_input` found in (task3_p_rsu.py).

    ###### P Task 3
    # TODO: Create an interprocess Queue for the processes listening for "rsu".
    rsu_queue = ...
    # TODO: Create a process with target `listen_for_input`.  These should
    # listen to the device "rsu". The rabbitmq_queue_name should match your
    # respective RabbitMQ queue name from "rsu" to "cav0". Use `rsu_queue` for
    # your interprocess queue.
    rsu_process = ...
    # TODO: Start the process.

    ###### P Task 3
```

```
27
28      ...
29
30      # make a loop periodic
31      images = set()
32      lidars = set()
33      cav0 = 0
34      try:
35          while True:
36              curr_time = time.time()
37              if next_time <= curr_time:
38                  ...
39
40                  fusion_output = local_fusion_wrapper(images, lidars)
41                  if fusion_output is not None:
42                      lf0 = TimedData(*fusion_output)
43
44                      ###### P Task 3: Part 2
45                      # TODO: Publish `lf0` to device "rsu" using the default
46                      # exchange ('') and your chosen name of the queue as the
47                      # routing_key. Call `encode_data(lf0)` for the body.
48
49                      # NOTE: uncomment if you want debugging
50                      # print(f'cav0 sending {lf0}')
51                      ###### P Task 3: Part 2
52
53                      # NOTE: Move to `task3_p_rsu.py` and finish Part 3.
54                  ...
```

Listing 37: The CAV's starting code for the Python NTWK Task of the SI app.

```
1  ###### P Task 3: Part 3
2  # This is the target function for a process listening to a RabbitMQ queue.
3  def listen_for_input(rabbitmq_queue_name, interprocess_queue: mp.Queue):
4      # TODO: initialize a RabbitMQ queue connection to `rabbitmq_queue_name`
5
6      # TODO: Create a callback function that should send data through the
7      # the interprocess queue paramater `interprocess_queue`
8      # TODO: Call `decode_data_bytes(...)` on the body in your callback
9
10     # TODO: Call `basic_consume` on `rabbitmq_queue_name` with your created
11     # callback function. Set `auto_ack` as True.
12
13     # TODO: start consuming.
14     pass
15
16
17 ###### P Task 3: Part 3
18 def rsu_main(start_time):
19     ...
20
```

```
    ###### P Task 3
    # TODO: Create two interprocess Queues for the processes listening for
    # "cav0" and "cav1".
    cav0_queue = ...
    cav1_queue = ...

    # TODO: Create two processes with target `listen_for_input`.
    # These should listen to the "cav0" and "cav1" device. The
    # rabbitmq_queue_name should match your respective RabbitMQ queue name per
    # CAV. Use `cav0_queue` and `cav1_queue` for your interprocess queue.
    cav0_process = ...
    cav1_process = ...
    # TODO: Start the processes.

    ###### P Task 3

    ###### P Task 3
    # TODO: initialize a RabbitMQ queue connection to device "cav0"

    ###### P Task 3

    ###### P Task 3
    # TODO: initialize a RabbitMQ queue connection to device "cav1"
    # Make sure the connection you create is different from the one you made
    # for "cav0".

    ###### P Task 3
    ...

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                fallthrough = curr_time + fallthrough_delay

                lf0_came = False
                lf1_came = False
                while time.time() < fallthrough and (not lf0_came
                                                     or not lf1_came):
                    lf0_came = safe_poll_TD_and_insert_TD(
                        cav0_queue, cav0_fusions, fallthrough) or lf0_came
                    lf1_came = safe_poll_TD_and_insert_TD(
                        cav1_queue, cav1_fusions, fallthrough) or lf1_came

                cav_lfs = extract_overlapping_data(cav0_fusions, cav1_fusions)
                if cav_lfs is not None:
                    cav0_fusion: TimedData
                    cav1_fusion: TimedData
                    cav0_fusion, cav1_fusion = cav_lfs
                    g_fusion = global_fusion(cav0_fusion.data,
                                             cav1_fusion.data)
                    intersect = cav0_fusion.interval.intersection(
```

```
                    cav1_fusion.interval).to_list()

                command_velocity_0 = calculate_angle(
                    g_fusion, cav0_fusion.data, cav_0)
                command_velocity_1 = calculate_angle(
                    g_fusion, cav1_fusion.data, cav_1)

                cv0 = TimedData(command_velocity_0, intersect)
                cv1 = TimedData(command_velocity_1, intersect)

                ###### P Task 3
                # TODO: Publish `cv0` to device "cav0" using the default
                # exchange ('') and your chosen name of the queue as the
                # routing_key. Call `encode_data(cv0)` for the body.

                # TODO: Publish `cv1` to "cav1" using the default exchange
                # ('') and your chosen name of the queue as the
                # routing_key. Call `encode_data(cv1)` for the body.

                # NOTE: uncomment if you would like debugging.
                # print(f'rsu sending {cv0} and {cv1}')
            ...
```

Listing 38: The RSU's starting code for the Python NTWK Task of the SI app.

You will implement the network capabilities between the CAV and the RSU. The CAV wants to send the output of `local_fusion` to the RSU for its call to `global_fusion`. The RSU then figures out each CAVs next direciton with `calculate_angle` and sends it back to them. The CAV then calls `command_motors` with that information.

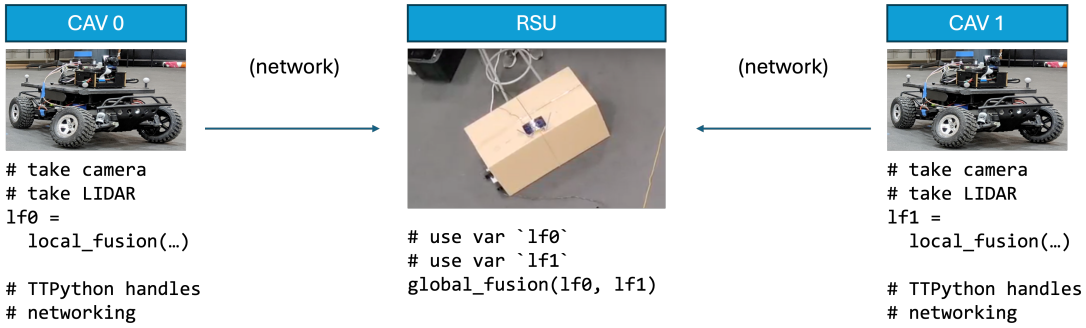


Figure A.15: System architecture describing CAV to RSU communication.

You will do the following:

- Specify that `"cav0"` and `"cav1"` will run sensor code and `local_fusion`.
- Specify that `"rsu"` will run `global_fusion` and `calculate_angle`.
- Specify that `"cav0"` and `"cav1"` will run `command_motors` from the RSU's data.

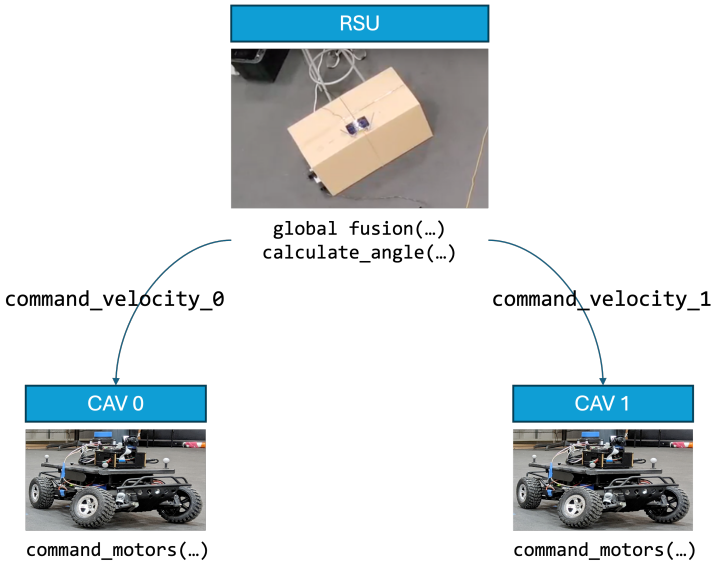


Figure A.16: System architecture describing RSU to CAV communication.

```
@GRAPHify
def smart_intersection(trigger):
    with TTClock.root() as root_clock:
        ...

        ###### T TASK 3:
        # Set up 3 devices: CAVs running `take_image`,
        #`take_LIDAR`, and `local_fusion`. The RSU runs `global_fusion` and
        #`calculate_angle`. Both CAVs receive their respective `velocity`
        # outputs to call `command_motors`.
        cav_0 = 0
        cav_1 = 1

        # NOTE: All SQs below should be assigned to a device.
        # TODO: Map the following SQs to run on "cav0".
        image0 = take_image(sample_window,
                            TTClock=root_clock,
                            TTPeriod=1_000_000,
                            TTPhase=0,
                            TTDataIntervalWidth=250_000)
        lidar0 = take_LIDAR(sample_window,
                            TTClock=root_clock,
                            TTPeriod=1_000_000,
                            TTPhase=0,
                            TTDataIntervalWidth=250_000)
        lf0 = local_fusion(image0, lidar0, cav_0)

        # TODO: Map the following SQs to run on "cav1".
        image1 = take_image(sample_window,
                            TTClock=root_clock,
```

```
                          TTPeriod=1_000_000,
                          TTPhase=0,
                          TTDataIntervalWidth=250_000)
        lidar1 = take_LIDAR(sample_window,
                          TTClock=root_clock,
                          TTPeriod=1_000_000,
                          TTPhase=0,
                          TTDataIntervalWidth=250_000)
        lf1 = local_fusion(image1, lidar1, cav_1)

        # TODO: Map the following to run on "rsu".
        gf = global_fusion(lf0, lf1)
        velocity_0 = calculate_angle(gf, lf0, cav_0)
        velocity_1 = calculate_angle(gf, lf1, cav_1)

        # TODO: Map the following to run on "cav0"
        final_result_0 = command_motors(velocity_0, cav_0)

        # TODO: Map the following to run on "cav1"
        final_result_1 = command_motors(velocity_1, cav_1)
```

Listing 39: The starting code for the TTPython NTWK Task of the SI app.

UF App

You will implement the network capabilities between the the optical camera, thermal camera, and the router. The optical image device will send optical images with EXIF data to the thermal camera. The thermal image device will send coregistered images from the optical camera device and its thermal image to the router.

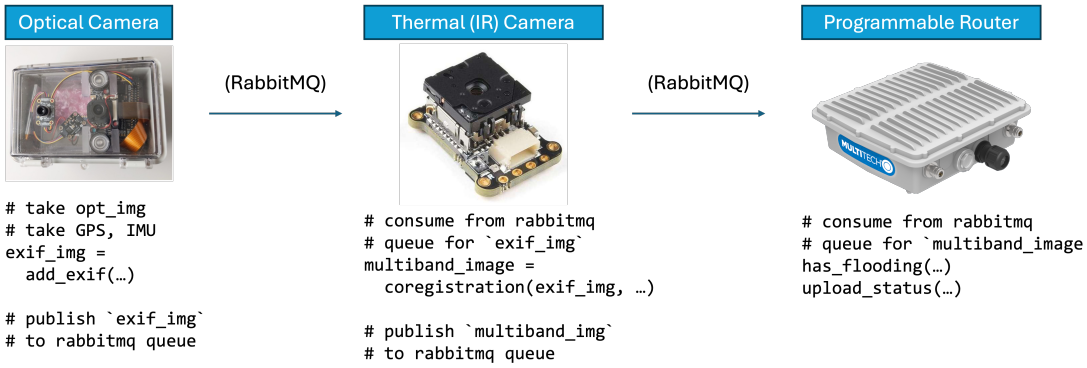


Figure A.17: System architecture describing the UF app.

You will be using RabbitMQ as a message broker to communicate between devices. Each device will create processes to listen for incoming data on the **network** queue. This is because RabbitMQ connections are blocking, and each device needs to run periodically. The process will take the data from the network and send it to the main process through an **interprocess** queue.

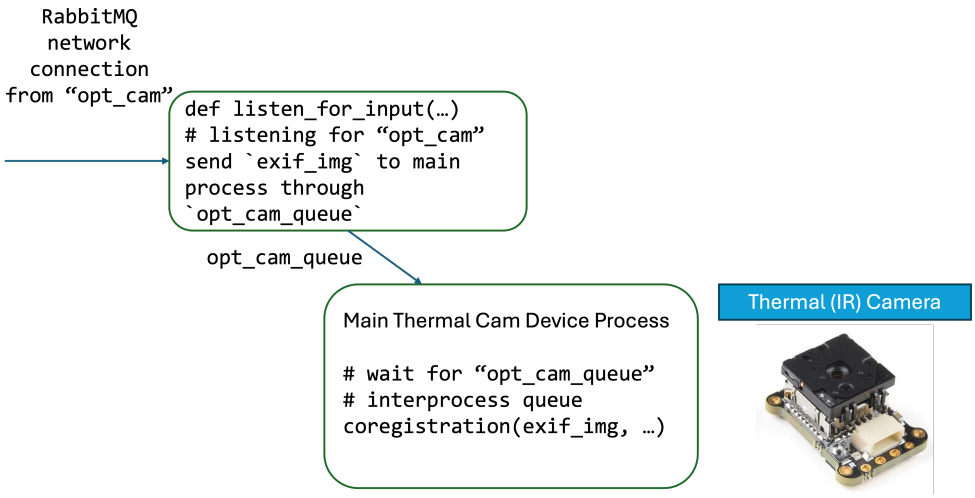


Figure A.18: System architecture describing RabbitMQ interface.

You will do the following:

- Initialize RabbitMQ connection on `"opt_cam"` and publish `exif_img` to `"ir_cam"`.
- Initialize RabbitMQ connection on `"ir_cam"`, receive from `"out_cam"` on RabbitMQ connection, set up processes and queues to read from RabbitMQ, and publish `ntwk_multiband_img` to `"router"`.

- Initialize RabbitMQ connection on `"router"`, receive from `"ir_cam"` on RabbitMQ connection, and set up processes and queues to read from RabbitMQ.

You will modify `task3_p_opt_cam.py`, `task3_p_ir_cam.py`, and `task3_p_router.py`. It is suggested you modify the files in this order. You can use the cell below to test your code.

```
def opt_cam_main(start_time):
    ...

    ###### P Task 3: Part 1
    # TODO: initialize a RabbitMQ queue connection to device "ir_cam"
    # As a reminder...
    # * instantiate a new pika blocking connection connected to localhost.
    # * Get the channel from the connection.
    # * Use the default exchange, declare the queue with a name of your choice.
    ###### P Task 3: Part 1

    ...

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                exif = add_exif_wrapper(opt_imgs, imus, gpss)

                if exif is not None:
                    exif_img = TimedData(*exif)
                    ###### P Task 3: Part 2
                    # TODO: Publish `exif_img` to "ir_cam" using the default
                    # exchange and your chosen name of the queue as the
                    # routing_key. Call `encode_data(exif_img)` for the body.

                    print(f'opt cam sending {exif_img}')
                    ###### P Task 3: Part 2
                ...
```

Listing 40: The optical camera's starting code for the TTPython NTWK Task of the UF app.

```
###### P Task 3: Part 3
# This is the target function for a process listening to a RabbitMQ queue.
def listen_for_input(rabbitmq_queue_name, interprocess_queue: mp.Queue):
    # TODO: initialize a RabbitMQ queue connection to `rabbitmq_queue_name`

    # TODO: Create a callback function that should send data through the
    # interprocess queue paramater `interprocess_queue`.
    # TODO: Call `decode_data_bytes(...)` on the body in your callback.

```

```
    # TODO: Call `basic_consume` on `rabbitmq_queue_name` with your created
    # callback function. Set `auto_ack` as True.

    # TODO: start consuming.
    pass

###### P Task 3: Part 3

def ir_cam_main(start_time):
    ...

    ###### P Task 3: Part 4
    # TODO: Create an interprocess Queue.
    opt_cam_queue = ...
    opt_cam_process = ...
    # TODO: Create a process with target `listen_for_input`.
    # This should listen to the "opt_cam" device. The rabbitmq_queue_name
    # should match your "opt_cam" RabbitMQ queue name. Use `opt_cam_queue`
    # for your interprocess queue.
    # TODO: Start the process.

    ###### P Task 3: Part 4

    ...

    ###### P Task 3: Part 5
    # TODO: initialize a RabbitMQ queue connection to device "router"
    ###### P Task 3: Part 5

    ...

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                multiband_img = coregistration_wrapper(exif_imgs,
                                                       thermal_imgs)

                if multiband_img is not None:
                    ntwk_multiband_img = TimedData(*multiband_img)
                    ###### P Task 3: Part 6
                    # TODO: Publish `ntwk_multiband_image` to "router" using
                    # the default exchange and your chosen name of the queue
                    # as the routing_key. Call
                    # `encode_data(ntwk_multiband_img)` for the body.

                    ###### P Task 3: Part 6
                    print(f'ir cam sent {ntwk_multiband_img}')
                ...
```

Listing 41: The thermal camera's starting code for the TTPython NTWK Task of the UF app.

```
def router_main(start_time):
    ...

    ###### P Task 3: Part 7
    # TODO: Create an interprocess Queue.
    thermal_cam_queue = ...
    thermal_cam_process = ...
    # TODO: Create and a process with target `listen_for_input`.
    # This should listen to the "ir_cam" device. The rabbitmq_queue_name
    # should match your "ir_cam" RabbitMQ queue name. Use
    # `thermal_cam_queue` for your interprocess queue.
    # TODO: Start the process.

    ###### P Task 3: Part 7

    ...

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                fallthrough = curr_time + fallthrough_delay

                multiband_img_came = False
                while time.time() < fallthrough and not multiband_img_came:
                    try:
                        multiband_img: TimedData = thermal_cam_queue.get(
                            block=False)
                        print('router heard back from thermal camera:
                              f'{multiband_img}')
                        multiband_img_came = True
                    except queue.Empty:
                        multiband_img = None

                if multiband_img_came:
                    classify = has_flooding(multiband_img.data)
                    upload_status(classify)
                ...
```

Listing 42: The router's starting code for the TTPython NTWK Task of the UF app.

You will implement the network capabilities between the the optical camera, thermal camera, and the router. The optical image device will send optical images with EXIF data to the thermal camera. The thermal image device will send coregistered images from the optical camera device and its thermal image to the router.

You will do the following:

- Specify that `"opt_camera"` will run `take_opt_image`, `take_gps`, `take_imu`, and `add_exif`.

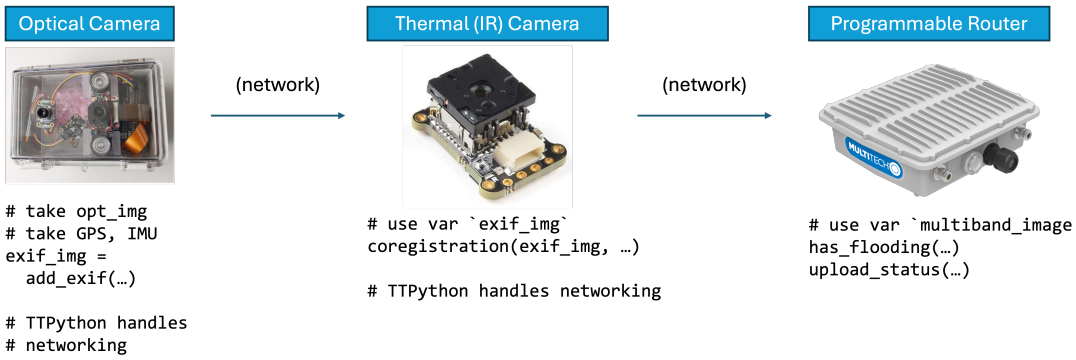


Figure A.19: System architecture describing the UF app.

- Specify that `"ir_camera"` will run `take_thermal_image` and `coregistration`.
- Specify that `"router"` will run `has_flooding` and `upload_status`.

```
@GRAPHify
def main(trigger):
    with TTClock.root() as root_clock:
        ...

        ###### T Task 3:
        # NOTE: All SQs below should be assigned to a device.
        # TODO: assign `take_opt_image`, `take_gps`, `take_imu`, and
        # `add_exif` to the device named "opt_camera".
        # TODO: Assign `take_thermal_image` and `coregistration` to device
        # named "ir_camera"
        lepton = take_thermal_image(sample_window,
                                    TTClock=root_clock,
                                    TTPeriod=2_000_000,
                                    TTPhase=0,
                                    TTDataIntervalWidth=500_000)

        images = take_opt_image(sample_window,
                                TTClock=root_clock,
                                TTPeriod=2_000_000,
                                TTPhase=0,
                                TTDataIntervalWidth=500_000)

        gps_exif = take_gps(sample_window,
                            TTClock=root_clock,
                            TTPeriod=2_000_000,
                            TTPhase=0,
                            TTDataIntervalWidth=500_000)

        imu_data = take_imu(sample_window,
                            TTClock=root_clock,
                            TTPeriod=2_000_000,
```

```
                            TTPhase=0,
                            TTDataIntervalWidth=500_000)

        exif_img = add_exif(images, gps_exif, imu_data)
        multiband_image = coregistration(exif_img, lepton)

        # TODO: Assign `has_flooding` and `upload_status` to the device named
        # "router".
        classify = has_flooding(multiband_image)
        uploaded = upload_status(classify)
```

Listing 43: The starting code for the TTPython NTWK Task of the UF app.

A.1.6 Task TTEH

SI App

You will implement a watchdog timer to handle time-triggered exception handling in the case of some action not happening by a specified deadline. In the smart intersection, `global_fusion` could still run even if a CAV becomes unresponsive for a while. You will write a time-triggered exception handler in `"rsu"` that runs `missing_input_fusion` to replace `"cav0"`'s `lf0` in the call `global_fusion`.

You will do the following:

- Call `missing_fusion_input` if deadline passes (`success` is **`False`** and `lf0_came` is **`False`**).
- Continue execution with the output of `missing_fusion_input` if the deadline passes.

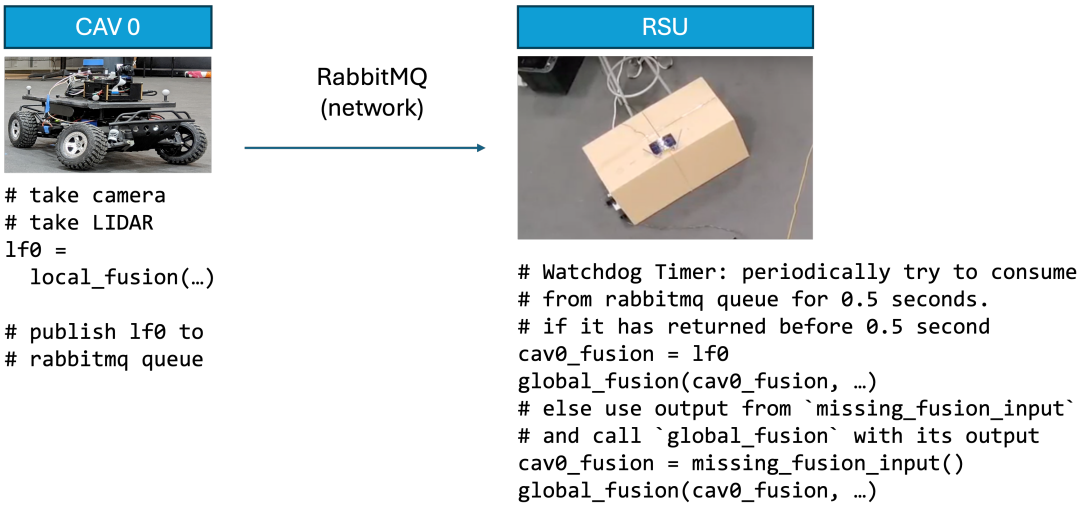


Figure A.20: System architecture describing the vanilla Python Time-Triggered Exception Handling for the RSU.

```
def rsu_main(start_time):
    ...

    try:
        # periodic control loop for the CAV
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                ###### T Task 4
                # NOTE: This is the normal path if everything comes in
                # on time. You don't need to do anything here.
                while time.time() < deadline_time and not success:
                    ...
```

```
16
17                  ###### T Task 4
18                  # This is the exceptional handling time. This should occur
19                  # if the "rsu" fails to synchronize data between both CAVs.
20                  # TODO: Replace the if statement checking if `success` is
21                  # False and `lf0_came` is False.
22                  if False:
23                      # TODO: Call the exceptional time handling recovery
24                      # function `missing_fusion_input(next_time)` to
25                      # replace `cav0_fusion`.
26
27                      # find cav1 match
28                      cav1_fusion = extract_fusion(cav1_fusions,
29                                                   cav0_fusion.interval)
30
31                      if cav1_fusion is not None:
32                          interval = cav1_fusion.interval
33
34                          # TODO: Call `global_fusion` with the replaced value
35                          # for its parameter `cav0_fusion.data` and
36                          # `cav1_fusion.data`. Use the `.data` field for
37                          # both arguments.
38
39                          # TODO: Call `calculate_angle` with `global_fusion`'s
40                          # output, the replaced value for its parameter
41                          # `cav0_fusion`, and `cav_0`. Use the `.data` field
42                          # for the replaced `cav0_fusion`.
43
44                          # This `calculate_angle` is done for "cav1"
45                          command_velocity_1 = calculate_angle(
46                              gf, cav1_fusion.data, cav_1)
47
48                          # TODO: Put the output of `calculate_angle` for
49                          # "cav0" here.
50                          cv0 = TimedData(..., interval.to_list())
51                          cv1 = TimedData(command_velocity_1, interval.to_list())
52
53                          # TODO: Send `cv0` to the device "cav0" over the
54                          # RabbitMQ connection. Look at its initializer above
55                          # for the RabbitMQ queue name. Remember to use
56                          # `encode_data(cv0)` on the body.
57
58                          # TODO: Send `cv1` to the device "cav1" over the
59                          # RabbitMQ connection. Look at its initializer above
60                          # for the RabbitMQ queue name. Remember to use
61                          # `encode_data(cv1)` on the body.
62
63                          # NOTE: You can uncomment this for debugging
64                          # print(f'rsu sent cv0:{cv0} and cv1:{cv1}')
65                  ...
```

Listing 44: The starting code for the Python TTEH Task of the SI app.

You will implement Plan B to handle time-triggered exception handling in the case of some action not happening by a specified deadline. In the smart intersection, `global_fusion` could still run even if a CAV becomes unresponsive for a while. You will write a Plan B for `"rsu"` that waits up to 0.5 seconds (`deadline_time`) for `"cav0"`'s data before running `global_fusion`. Specify that Plan B will replace `"cav0"`'s local fusion data with the Plan B handler `missing_fusion_input`. Thus, `global_fusion` should still fire periodically either with `"cav0"`'s data or from the output of `missing_fusion_input`.

You will do the following:

- Call `missing_fusion_input` if `lf0` not received by `deadline_time`.
- Continue execution with the output of `missing_fusion_input` if Plan B occurs.

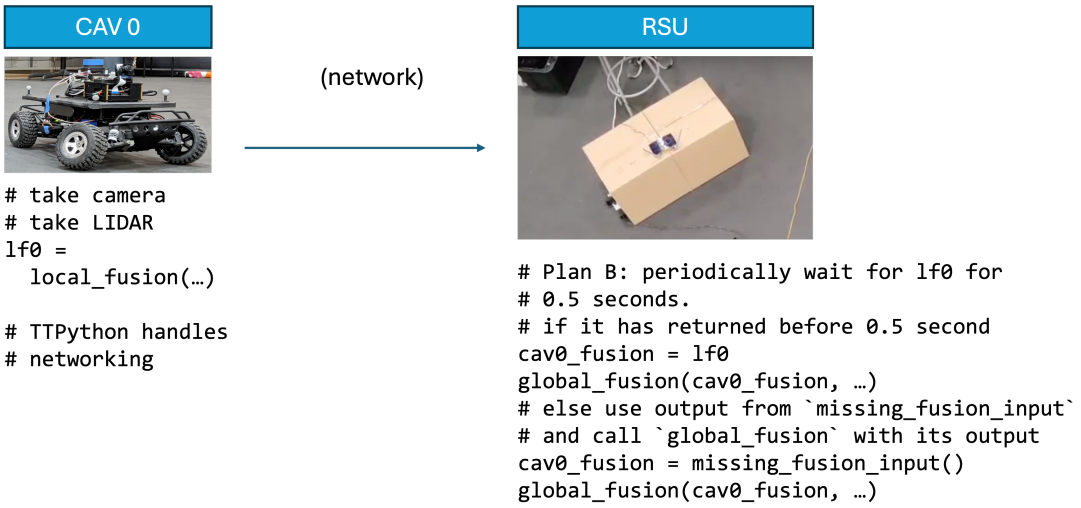


Figure A.21: System architecture describing the TTPython Time-Triggered Exception Handling for the RSU.

```
@SQify
def missing_fusion_input():
    print('running plan B!')
    return None

# T Task 4:
@GRAPHify
def cav_to_rsu_fusion(trigger):
    with TTClock.root() as root_clock:
        ...

        with TTConstraint(name="cav0"):
            ...
            lf0 = local_fusion(image_0, lidar_0, cav_0)

        with TTConstraint(name="cav1"):
```

```
            ...

        # T Task 4
        with TTConstraint(name="rsu"):
            local_deadline = identity(cav_0,
                                      TTClock=root_clock,
                                      TTPeriod=1_000_000,
                                      TTPhase=0,
                                      TTDataIntervalWidth=250_000)

            ###### T Task 4:
            # TODO: Call the Plan B recovery function `missing_fusion_input`
            # for the `lf0` data before `global_fusion` runs if the RSU does
            # not hear back from cav0 within 500_000 microseconds. This
            # 500_000 deadline is given below. We still want to run global
            # fusion with a replaced value for `lf0`.
            deadline_time = READ_TTCLOCK(local_deadline,
                                         TTClock=root_clock) + 500_000

            # TODO: Use the new var checking for `lf0`. We want to
            # still run `global_fusion` even if `lf0` is missing.
            gf = global_fusion(lf0, lf1)
            velocity_0 = calculate_angle(gf, lf0, cav_0)
            velocity_1 = calculate_angle(gf, lf1, cav_1)

        with TTConstraint(name="cav0"):
            final_result_0 = command_motors(velocity_0, cav_0)

        with TTConstraint(name="cav1"):
            final_result_1 = command_motors(velocity_1, cav_1)
```

Listing 45: The starting code for the TTPython TTEH Task of the SI app.

UF App

You will implement a watchdog timer to handle time-triggered exception handling in the case of some action not happening by a specified deadline. The thermal camera is flaky and can become unresponsive, so we set a deadline that if reached, the code should call `reset_cam`. You will inform the `"router"` that the thermal camera is unresponsive by sending it the output of the call to `reset_cam`.

Figure A.22: System architecture describing the vanilla Python Time-Triggered Exception Handling for the thermal camera.

You will do the following:

- Call `reset_cam` if the deadline passes (`thermal_image_came` is **`False`** and `success` is **`False`**).
- Send the output of `reset_cam` to the `"router"` if the deadline passes.

```
def ir_cam_main(start_time):
    ...

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...
                success = False
                ###### T Task 4
                # NOTE: This is the normal path if everything comes in on
                # time. You don't need to do anything here.
                while time.time() < deadline and not success:
                    ...

                ###### T Task 4
```

```
                # TODO: Replace the if statement checking if
                # `thermal_img_came` and `success` is False.
                if False:
                    exif_img_td = extract_data(exif_imgs,
                                               Interval(0, time.time()))
                    # TODO: Call the exceptional time handling recovery
                    # function `reset_cam(next_time)` to replace
                    # `thermal_img`.

                    if exif_img_td is not None:
                        exif_img = exif_img_td.data
                        interval = exif_img_td.interval

                        # TODO: call `coregistration` with `exif_data`
                        # and the replaced value for its parameter
                        # `thermal_img`. Remember to use its `.data`
                        # field, like how `exif_img` is created above.

                        # TODO: put the output of `coregistration` here.
                        ntwk_multiband_img = TimedData(...,
                                                       interval.to_list())

                        # TODO: Send `ntwk_multiband_img` to the device
                        # "router" over the RabbitMQ connection. Look at
                        # its initializer above for the RabbitMQ queue
                        # name. Remember to use
                        # `encode_data(ntwk_multiband_img)` on the body.

                        print(f'ir cam sent {ntwk_multiband_img}')
                ###### T Task 4
                ...
```

Listing 46: The starting code for the Python TTEH Task of the UF app.

You will implement Plan B to handle time-triggered exception handling in the case of some action not happening by a specified deadline. The thermal camera is flaky and can become unresponsive, so we set a deadline that if reached, the code should call `reset_cam`. You will inform the `"router"` that the thermal camera is unresponsive by sending it the output of the call to `reset_cam`.

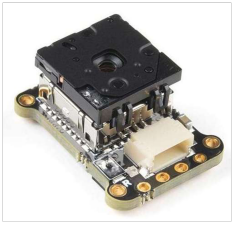
You will do the following:

- Call `reset_cam` if `thermal_img` not received by `deadline_time`.
- Continue execution with the output of `reset_cam` if Plan B occurs.

```
@SQify
def reset_cam():
    import time

    print('Thermal Camera deadline reached: resetting...')
    time.sleep(0.5)
    print('done')
```

Thermal (IR) Camera



```
# Plan B: periodically wait for thermal
# img for 1 second.
success_img = take_thermal_img()
# if it has returned before 1 second
coregistration(exif_img, …)
# else run `reset_cam` and call coreg with
# its value
replaced_val = reset_cam()
coregistration(replaced_val, …)
```

Figure A.23: System architecture describing the TTPython Time-Triggered Exception Handling for the thermal camera.

```
    return "Thermal Camera failed to respond"

@GRAPHify
def main(trigger):
    with TTClock.root() as root_clock:
        ...

        with TTConstraint(name="ir_camera"):
            t_img_start_time, thermal_img = take_buggy_image(
                sample_window,
                TTClock=root_clock,
                TTPeriod=2_000_000,
                TTPhase=0,
                TTDataIntervalWidth=500_000)

            ###### T Task 4:
            # TODO: Call the Plan B `reset_cam()` for `thermal_img` before
            # `coregistration` if function `take_buggy_image` does not return
            # within `deadline_time`. Make sure that `coregistration` still
            # runs if Plan B fires.
            deadline_time = READ_TTCLOCK(t_img_start_time,
                                         TTClock=root_clock) + 1_000_000

            # TODO: replace `thermal_img` with output of the Plan B handler.
            # Make sure that `coregistration` still runs if Plan B fires.
            multiband_image = coregistration(exif_img, thermal_img)

        with TTConstraint(name="router"):
            classify = has_flooding(multiband_image)
```

```
37              uploaded = upload_status(classify)
```

Listing 47: The starting code for the TTPython TTEH Task of the UF app.

A.1.7 Task CE

SI App

You will modify the existing code of the smart intersection. The current architecture is shown below and split across two diagrams for clarity of code execution order.

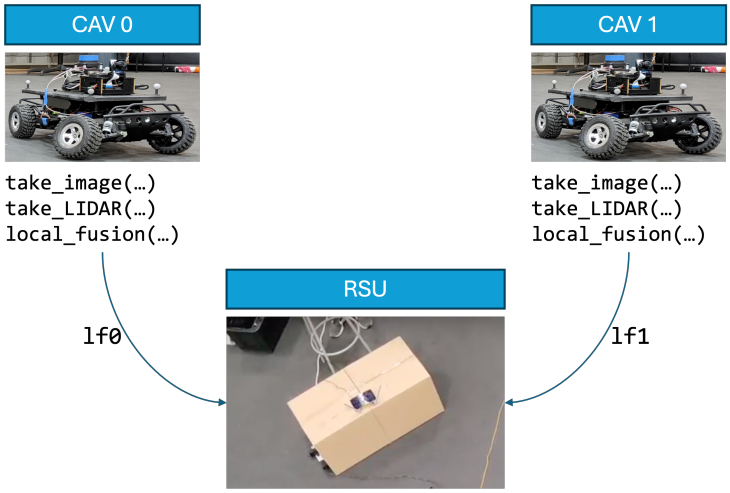


Figure A.24: System architecture describing CAV to RSU communication.

The CAVs first generates LIDAR and camera information and synchronizes them in `local_fusion`. It sends this to the RSU. The RSU call `global_fusion` with info from both CAVs. It plans routing for each CAV with `calculate_angle` and sends it to each.

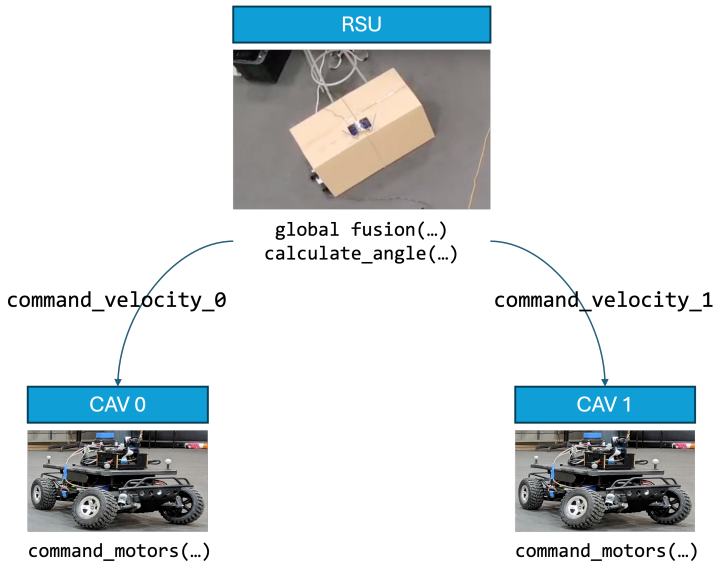


Figure A.25: System architecture describing RSU to CAV communication.

To make the cars more autonomous, `calculate_angle` can be done on the car. Ensure that when moving `calculate_angle` to the CAVs that the corresponding exception handler is also changed as

well.

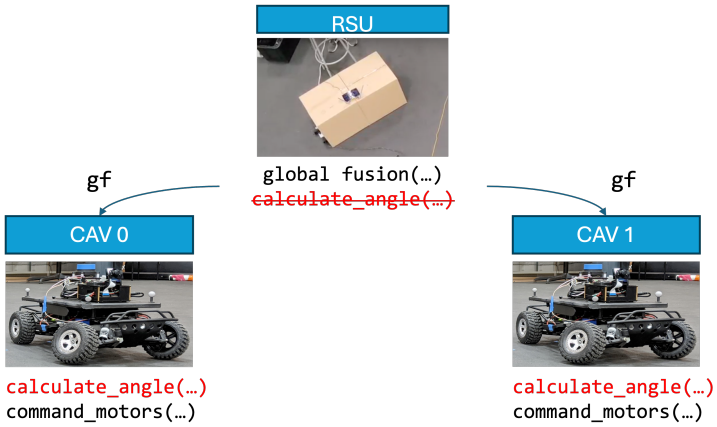


Figure A.26: CE task system architecture change for the SI app.

You will do the following:

- Move the functions `calculate_angle` for each CAV to their respective file (`calculate_angle` for CAV0 to `"task5_p_cav0.py"`). The `"rsu"` sends `gf` instead.
- Modify each CAV to listen for `gf` instead. Modify the exceptional time handling code to call `no_global_fusion` instead.

You will modify `task5_p_rsu.py`, `task5_p_cav0.py`, and `task5_p_cav1.py`. You can use the cell below to test your code.

```
def rsu_main(start_time):
    ...

    try:
        # periodic control loop for the CAV
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                ###### P Task 5
                # TODO: uncomment me
                # ntwk_gf = TimedData(gf, intersect)

                # skips odd iterations to introduce failure in the system
                if cav0_fusion is not None and cav0_fusion.data[0] % 2 == 0:

                    ###### P Task 5:
                    # TODO: Move `calculate_angle` to "cav0"
                    # (task5_p_cav0.py) and "cav1" (task5_p_cav1.py).
                    command_velocity_0 = calculate_angle(
```

```
                        gf, cav0_fusion.data, cav_0)
                    command_velocity_1 = calculate_angle(
                        gf, cav1_fusion.data, cav_1)

                    # TODO: Remove the follownig code.
                    cv0 = TimedData(command_velocity_0, intersect)
                    cv1 = TimedData(command_velocity_1, intersect)

                    # TODO: Change `encode_data(...)` to
                    # `encode_data(ntwk_gf)`
                    channel0.basic_publish(exchange='',
                                           routing_key=queue_name0,
                                           body=encode_data(cv0))
                    channel1.basic_publish(exchange='',
                                           routing_key=queue_name1,
                                           body=encode_data(cv1))
                    ###### P Task 5

                else:
                    ...
```

Listing 48: The RSU's starting code for the Python CE Task of the SI app.

```
def cav_main0(start_time):
    ...

    ###### P Task 5
    # TODO: Rename `velocities` to `gfs`
    velocities = set()
    ###### P Task 5

    ...

    try:
        # periodic control loop for the CAV
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                ...

                # listening to the RSU
                moving_success = False
                while time.time() < rsu_deadline and not moving_success:
                    ###### P Task 5
                    # TODO: Rename the var `velocities` below to `gfs`.
                    # TODO: Rename `command_velocity` to `gf`.
                    safe_poll_TD_and_insert_TD(rsu_queue,
                                               velocities, rsu_deadline)
                    command_velocity = extract_data(
                        velocities, Interval(start_time, rsu_deadline))
                    if command_velocity is not None:
```

```
                moving_success = True

        if not moving_success:
            ###### P Task 5
            # TODO: Change `emergency_stop(...)` to
            # `no_global_fusion(start_time)`
            # TODO: Rename `command_velocity` to gf
            command_velocity = emergency_stop(start_time)
            # print('cav0 emergency stop')

        if lf0 is None:
            print(f'catastrophic error for cav0')
        else:

            ###### P Task 5
            # TODO: Move `calculate_angle` from (task5_p_rsu.py) here
            # use `lf0` instead of the cav{cav_num}_fusion. Use
            # `gf.data` and `lf0.data` for its arguments.

            # TODO: Use `command_velocity` instead of its `.data`
            # field once changes above have been made.
            # This should use the output of `calculate_angle`.
            command_motors(command_velocity.data, cav_0)
            ###### P Task 5
        ...
```

Listing 49: The CAV's starting code for the Python CE Task of the SI app.

You will modify the existing code of the smart intersection. The current architecture is shown below and split across two diagrams for clarity of code execution order.

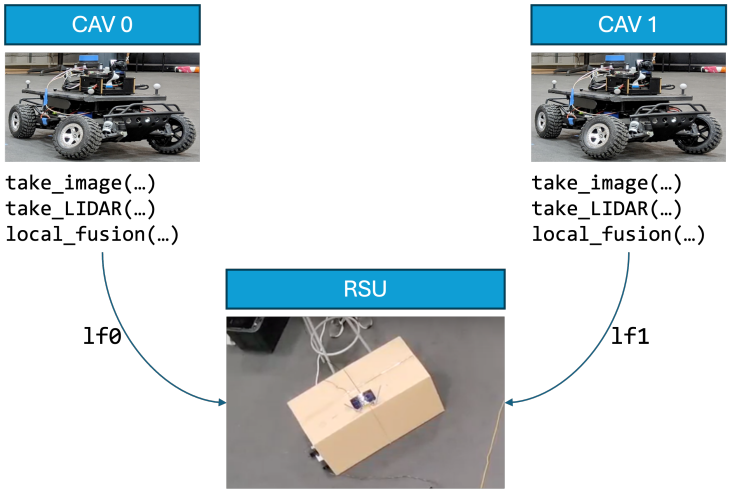


Figure A.27: System architecture describing CAV to RSU communication.

The CAVs first generates LIDAR and camera information and synchronizes them in `local_fusion`. It sends this to the RSU. The RSU call `global_fusion` with info from both CAVs. It plans routing for each CAV with `calculate_angle` and sends it to each.

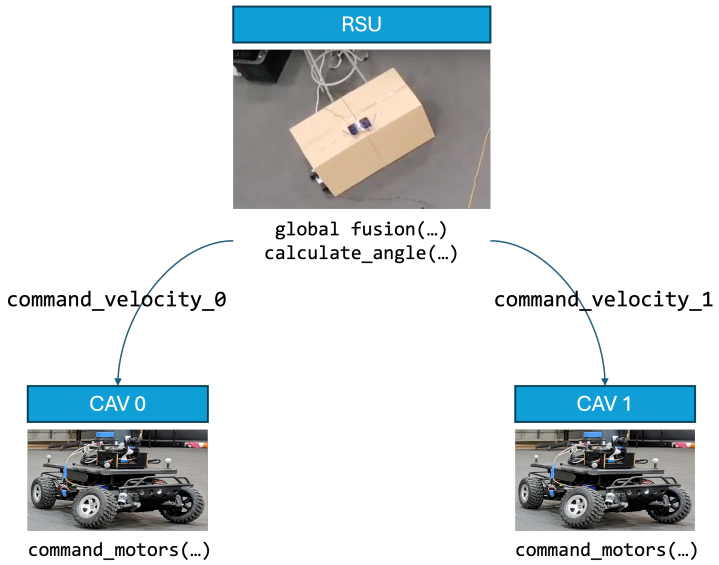


Figure A.28: System architecture describing RSU to CAV communication.

To make the cars more autonomous, `calculate_angle` can be done on the car. Ensure that when moving `calculate_angle` to the CAVs that the corresponding Plan B is also changed as well.

You will do the following:

- Move the respective SQ `calculate_angle` for each device onto their respective device.

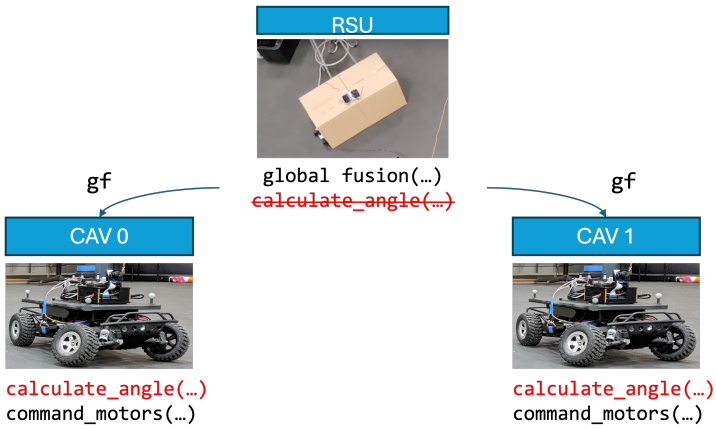


Figure A.29: CE task system architecture change for the SI app.

- Change Plan B to work with `gf` instead of `command_velocity_##`.

```
@GRAPHify
def smart_intersection(trigger):
    with TTClock.root() as root_clock:
        ...

        with TTConstraint(name="rsu"):
            cav_0_fusion = TTFinishByOtherwise(lf0,
                                               TTTimeDeadline=deadline,
                                               TTPlanB=missing_fusion_input(),
                                               TTWillContinue=True)

            cav_1_fusion = TTFinishByOtherwise(lf1,
                                               TTTimeDeadline=deadline,
                                               TTPlanB=missing_fusion_input(),
                                               TTWillContinue=True)

            gf = global_fusion(cav_0_fusion, cav_1_fusion)

            ###### T Task 5
            # TODO: Move each `calculate_angle` for "cav0" and "cav1" to
            # run on its respective device "cav0" and "cav1".
            command_velocity_0 = calculate_angle(gf, lf0, cav_0)
            command_velocity_1 = calculate_angle(gf, lf1, cav_1)

        with TTConstraint(name="cav0"):
            # TODO: this Plan B handler safeguards against network
            # transmission of `command_velocity_0`. Once moving
            # `calculate_angle`, this should safeguard against the
            # transmission of `gf`. Change the Plan B function to
            # `no_global_fusion`. The other parameters will stay the same.
            velocity_0 = TTFinishByOtherwise(command_velocity_0,
                                             TTTimeDeadline=cav_0_deadline,
```

```
                                            TTPlanB=emergency_stop(),
                                            TTWillContinue=True)
        final_result_0 = command_motors(velocity_0, cav_0)

    with TTConstraint(name="cav1"):
        # TODO: this Plan B handler safeguards against network
        # transmission of `command_velocity_1`. Once moving
        # `calculate_angle`, this should safeguard against the
        # transmission of `gf`. Change the Plan B function to
        # `no_global_fusion`. The other parameters will stay the same.
        velocity_1 = TTFinishByOtherwise(command_velocity_1,
                                            TTTimeDeadline=cav_1_deadline,
                                            TTPlanB=emergency_stop(),
                                            TTWillContinue=True)
        final_result_1 = command_motors(velocity_1, cav_1)
```

Listing 50: The starting code for the TTPython CE Task of the SI app.

UF App

You will modify the existing code of the urban flooding observation application.

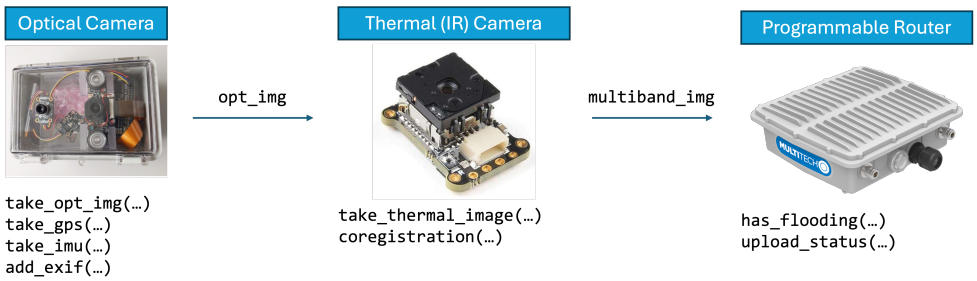


Figure A.30: System architecture describing the UF app.

`"opt_cam"` device takes an image and combine this with the GPS and IMU data in `add_exif`. It sends this to `"thermal_cam"` device, which calls `coregistration` with it and a thermal image. This is then sent to the device `"router"` which calls `has_flooding` on it and `upload_status`.

In this task, we will move `coregistration` (on `"thermal_cam"`) to `"router"`, as it has better computing power. Device `"opt_cam"` should send its `exif_img` to the `"router"` instead. Device `"thermal_cam"` will now send its `thermal_img`.

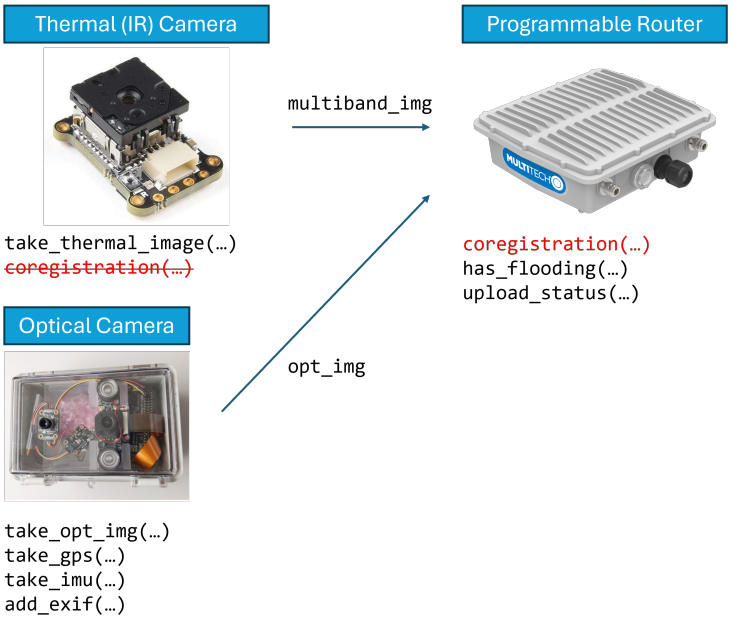


Figure A.31: CE task system architecture change for the UF app.

You will do the following:

- Change where `"opt_cam"` is sending `exif_img` to (`"router"` device).
- Move the function `coregistration` to `"router"`. The `"thermal_cam"` sends `thermal_img` instead.

- Create a process listening for `"opt_cam"`'s RabbitMQ network queue connection on `"router"`.

```
def opt_cam_main(start_time):
    ...

    # Create RabbitMQ connection
    connection = pika.BlockingConnection(
        pika.ConnectionParameters('localhost'))
    channel = connection.channel()

    ###### P Task 5
    # TODO: Send this to the "router" instead of "ir_cam"
    queue_name = 'opt_to_ir'
    channel.queue_declare(queue=queue_name)
    ###### P Task 5

    ...
```

Listing 51: The optical camera's starting code for the Python CE Task of the UF app.

```
def ir_cam_main(start_time):
    ...

    ### Listen to optical camera on RabbitMQ.
    ###### P Task 5
    # TODO: Comment/Delete this code since `opt_cam` doesn't talk to `ir_cam`
    # anymore.
    opt_cam_queue = mp.Queue()
    opt_cam_process = mp.Process(target=listen_for_input,
                                 args=('opt_to_ir',
                                       opt_cam_queue))
    opt_cam_process.start()
    exif_imgs = set()
    ######

    ...

    ###### P Task 5
    # TODO: Comment/Delete this code since `opt_cam` doesn't talk to `ir_cam`
    # anymore.
    remove_old_data_and_insert(opt_cam_queue, exif_imgs, start_time)
    ###### P Task 5

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                deadline = curr_time + deadline_offset

```

```
            thermal_img_came = False
            success = False
            ###### P Task 5
            # TODO: Comment/Delete this code since `opt_cam` doesn't
            # talk to `ir_cam` anymore.
            exif_img_came = False
            while time.time() < deadline and not success:

                ###### P Task 5
                # TODO: Comment/Delete this code since `opt_cam` doesn't
                # talk to `ir_cam` anymore.
                exif_img_came = safe_poll_TD_and_insert_TD(
                    opt_cam_queue, exif_imgs,
                    deadline) or exif_img_came
                ###### P Task 5

                thermal_img_came = safe_poll_and_insert_TD(
                    thermal_cam_queue, thermal_imgs,
                    deadline) or thermal_img_came

                # synchronize between exif and thermal imgs
                ###### P Task 5:
                # TODO: This synchronization code between `exif_imgs` and
                # `thermal_imgs` needs to move to "task5_p_router.py".
                # Most of the code structure mimics what you will do in
                # "task5_p_router.py". After modification, none of this
                # code should be here. ------------------------>
                data_pair = extract_overlapping_data_pair(
                    exif_imgs, thermal_imgs)

                # get a synchronized pair of `exif_img` and `opt_img`.
                if data_pair is not None:
                    opt_img: TimedData
                    thermal_img: TimedData
                    opt_img, thermal_img = data_pair

                    ###### P Task 5
                    # TODO: Copy this function `coregistration` to the
                    # "router" device.
                    multiband_img = coregistration(opt_img.data,
                                                   thermal_img.data)
                    ######
                    interval = opt_img.interval.intersection(
                        thermal_img.interval).to_list()
                else:
                    multiband_img = None
                ###### <--------------------
                ###### Code above should be removed/moved.

                ###### P Task 5:
                # TODO: Uncomment the following code to get
                # a `thermal_img`.
```

```
                    # thermal_img = extract_data(thermal_imgs,
                    #                            Interval(0, time.time()))

                    # TODO: Change the flag to `thermal_img` instead of
                    # `multiband_img`.
                    if multiband_img is not None:
                        success = True

                        # TODO: Remove this line
                        ntwk_multiband_img = TimedData(multiband_img,
                                                       interval)

                        # TODO: Change the body to
                        # 'encode_data(thermal_img)'.
                        channel.basic_publish(
                            exchange='',
                            routing_key=router_queue_name,
                            body=encode_data(ntwk_multiband_img))
                        print(f'ir cam sent {multiband_img}')
                    #######

                # Exceptional handling if thermal cam is unresponsive
                if not success and not thermal_img_came:
                    exif_img_td = extract_data(exif_imgs,
                                               Interval(start_time,
                                                        time.time()))
                    response = reset_cam(start_time)

                    ###### P Task 5
                    # TODO: Remove following code
                    multiband_img = coregistration(exif_img_td.data,
                                                   response.data)
                    interval = exif_img_td.interval
                    ntwk_multiband_img = TimedData(multiband_img,
                                                   interval.to_list())

                    # TODO: Change `encode_data(...)` to
                    # `encode_data(response)`
                    channel.basic_publish(
                        exchange='',
                        routing_key=router_queue_name,
                        body=encode_data(ntwk_multiband_img))
                    print(f'ir cam sent issue:{ntwk_multiband_img}')
                    ###### P Task 5
                ...
```

Listing 52: The optical camera's starting code for the Python CE Task of the UF app.

```
def router_main(start_time):
    ...

```

```
    ###### P Task 5
    opt_cam_queue = mp.Queue()
    # opt_cam_process = ...
    # TODO: Listen to "opt_cam" with a separate Process. Set the target as
    # `listen_for_input`, and args as the rabbitmq_queue_name specified in
    # `task5_opt_cam.py` and the interprocess queue as `opt_cam_queue`.
    # It should look similar to listening to "ir_cam".

    ######

    ...

    exif_imgs = set()
    ###### P Task 5
    # TODO: Rename `multiband_imgs` to `thermal_imgs`
    multiband_imgs = set()

    try:
        while True:
            curr_time = time.time()
            if next_time <= curr_time:
                deadline = curr_time + deadline_offset

                ######  P Task 5
                # TODO: Rename `multiband_img_came` to `thermal_img_came`.
                multiband_img_came = False
                exif_img_came = False
                # TODO: Change the second clause of `while` check to include
                # exif_img_came.
                # `and (not multiband_img_came or not exif_img_came)`
                while time.time() < deadline and not multiband_img_came:
                    exif_img_came = safe_poll_TD_and_insert_TD(
                        opt_cam_queue, exif_imgs, deadline) or exif_img_came
                    multiband_img_came = safe_poll_TD_and_insert_TD(
                        thermal_cam_queue, multiband_imgs,
                        deadline) or multiband_img_came

                ###### P Task 5
                # TODO: Synchronize a `exif_img` and `thermal_img` with
                # `extract_overlapping_data_pair(exif_imgs, thermal_imgs)`

                # TODO: Replace if statement with check if
                # `extract_..._pair` call above is not None
                if multiband_img_came:
                    # TODO: uncomment and assign a synchronized `opt_img` and
                    # `thermal_img` from the call above. The return should be
                    # a tuple.
                    # opt_img = ...
                    # thermal_img = ...

                    # TODO: delete/comment below
                    multiband_img = extract_data(multiband_imgs,
```

```
                                                    Interval(start_time,
                                                             time.time())).data

                    # TODO: move `coregistration` into this body of code

                    classify = has_flooding(multiband_img)
                    upload_status(classify)

                ...
```

Listing 53: The router's starting code for the Python CE Task of the UF app.

You will modify the existing code of the urban flooding observation application.

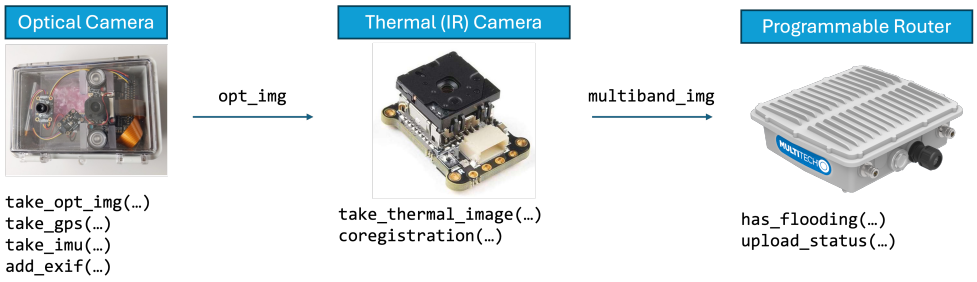


Figure A.32: System architecture describing the UF app.

`"opt_cam"` device takes an image and combine this with the GPS and IMU data in `add_exif`. It sends this to `"thermal_cam"` device, which calls `coregistration` with it and a thermal image. This is then sent to the device `"router"` which calls `has_flooding` on it and `upload_status`.

In this task, we will move `coregistration` (on `"thermal_cam"`) to `"router"`, as it has better computing power. Device `"opt_cam"` should send its `exif_img` to the `"router"` instead. Device `"thermal_cam"` will now send its `thermal_img`.

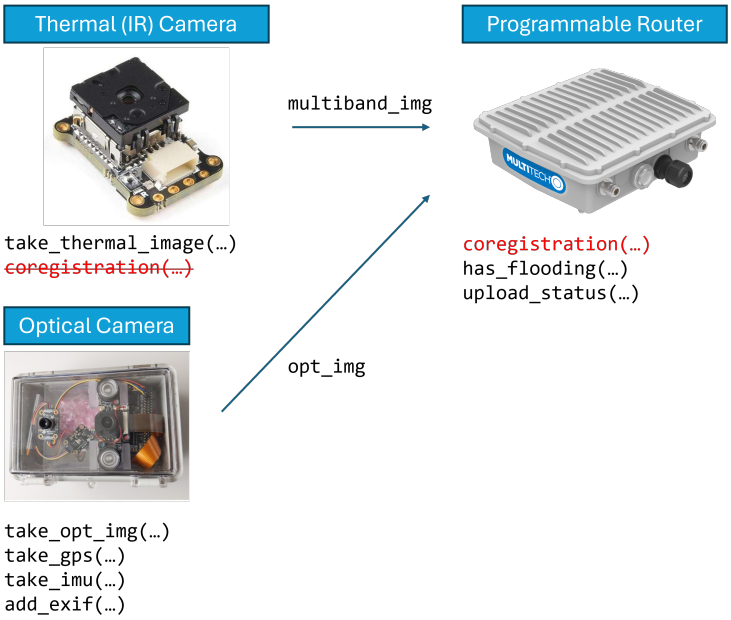


Figure A.33: CE task system architecture change for the UF app.

You will do the following:

- Move ‘coregistration‘ to device ‘”router”‘.

```
1 @GRAPHify
2 def main(trigger):
```

```
    with TTClock.root() as root_clock:
        ...

        with TTConstraint(name="ir_camera"):
            t_img_start_time, thermal_image = take_buggy_image(
                sample_window,
                TTClock=root_clock,
                TTPeriod=2_000_000,
                TTPhase=0,
                TTDataIntervalWidth=500_000)
            deadline = READ_TTCLOCK(t_img_start_time,
                                    TTClock=root_clock) + 1_000_000
            successful_img = TTFinishByOtherwise(thermal_image,
                                                 TTTimeDeadline=deadline,
                                                 TTPlanB=reset_cam(),
                                                 TTWillContinue=True)
            ###### T Task 5:
            # TODO: Move `coregistration` to the device "router".
            multiband_image = coregistration(exif_img, successful_img)

        with TTConstraint(name="router"):
            classify = has_flooding(multiband_image)
            uploaded = upload_status(classify)
```

Listing 54: The starting code for the TTPython CE Task of the UF app.

A.1.8 TTPython/Python Questionnaire

The following questions were rated by participants from either a 1-5 or a 1-10 scale.

- I think that I would like to use this system frequently.
- I found the system unnecessarily complex.
- I thought the system was easy to use.
- I think that I would need the support of a technical person to be able to use this system.
- I found the various functions in this system were well-integrated.
- I thought there was too much inconsistency in this system.
- I would imagine that most people would learn to use this system very quickly.
- I found the system very cumbersome to use.
- I feel very confident using the system.
- I needed to learn a lot of things before I could get going with this system.
- How mentally demanding was the task?
- How physically demanding was the task?

- How hurried or rushed was the pace of the task?
- How successful were you in accomplishing what you were asked to do?
- How hard did you have to work to accomplish your level of performance?

Bibliography

[1] Grant G21AP10626 Cornell University — highergov.com. `www.highergov.com/grant/G21AP10626/`, 2018. 3.2

[2] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2019. 5.1

[3] Bilal Akil, Ying Zhou, and Uwe Röhm. On the usability of Hadoop MapReduce, Apache Spark & Apache flink for data science. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 303–310, 2017. 5.2

[4] Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. *PRET-C: A new language for programming precision timed architectures*. PhD thesis, INRIA, 2009. 5.1

[5] Fatima Anwar, Sandeep D'souza, Andrew Symington, Adwait Dongare, Ragunathan Rajkumar, Anthony Rowe, and Mani Srivastava. Timeline: An operating system abstraction for time-aware applications. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 191–202. IEEE, 2016. 5.1

[6] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. *ACM SIGOPS Operating Systems Review*, 41(3):159–172, 2007. 1.1, 5.1

[7] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services*, EESR '05, page 19–24. USENIX Association, 2005. 1.1, 5.1

[8] Frédéric Boussinot and Robert De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79 (9):1293–1304, 1991. 5.1

[9] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 3.2.2

[10] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. 4.1.9

[11] John Brooke. SUS: A quick and dirty usability scale. *Usability Eval. Ind.*, 189, 1995. (document), 4.1.7, 4.7.1, 4.1, 5.2

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 519–538. Association for Computing Machinery, 2005. 5.1

[13] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. Glacier: transitive class immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 496–506. IEEE Press, 2017. 5.2

[14] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in Obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. 5.2

[15] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput.-Hum. Interact.*, 28(4), 2021. 5.2

[16] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer Berlin Heidelberg, 2007. 5.1

[17] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3): 1–22, 2013. 5.1

[18] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18. Association for Computing Machinery, 2003. 5.1

[19] Matthew C. Davis, Emad Aghayi, Thomas D. Latoza, Xiaoyin Wang, Brad A. Myers, and Joshua Sunshine. What's (Not) Working in Programmer User Studies? *ACM Trans. Softw. Eng. Methodol.*, 32(5), 2023. 5.2

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. 5.1

[21] Jack B. Dennis and David P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ISCA '75, page 126–132. Association for Computing Machinery, 1974. 5.1

[22] Alexandre Donzé. On signal temporal logic. In *International Conference on Runtime Verification*, pages 382–383. Springer, 2013. 5.1

[23] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, 1997. 5.1

[24] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. Usability Barriers for Liquid Types. *Proc. ACM Program. Lang.*, 9(PLDI), 2025. 5.2

[25] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 1–11. Association for Computing Machinery, 2003. 5.1

[26] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 1–2. Association for Computing Machinery, 2005. 5.1

[27] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. 5.1

[28] Sandra G. Hart and Lowell E. Staveland. Development of NASA-TLX (Task Load Index): Results of

Empirical and Theoretical Research. In Peter A. Hancock and Najmedin Meshkati, editors, *Human Mental Workload*, volume 52 of *Advances in Psychology*, pages 139–183. North-Holland, 1988. (document), 4.1.7, 4.7.1, 4.1

[29] Timothy W Hnat, Tamim I Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. MacroLab: a vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, 2008. 5.1

[30] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11): 1920–1930, 2008. 5.2

[31] Mohammad Khayatian, Rachel Dedinsky, Sarthake Choudhary, Mohammadreza Mehrabian, and Aviral Shrivastava. R2IM – Robust and Resilient Intersection Management of Connected Autonomous Vehicles. In *proceedings of The 23rd IEEE International Conference on Intelligent Transportation Systems*, 2020. 3.1

[32] Mohammad Khayatian, Mohammadreza Mehrabian, Edward Andert, Rachel Dedinsky, Sarthake Choudhary, Yingyan Lou, and Aviral Shirvastava. A Survey on Intersection Management of Connected Autonomous Vehicles. *ACM Trans. Cyber-Phys. Syst.*, 4(4), 2020. 1.1, 3.1

[33] Mohammad Khayatian, Mohammadreza Mehrabian, Edward Andert, Reese Grimsley, Kyle Liang, Yi Hu, Ian McCormack, Carlee Joe-Wong, Jonathan Aldrich, Bob Iannucci, and Aviral Shrivastava. Plan B - Design Methodology for Cyber-Physical Systems Robust to Timing Failures. *ACM Trans. Cyber-Phys. Syst.*, 6(3), 2022. 1.1, 2.2.2

[34] Ben Kluwe and Luke Van Horn. Lepton 3 port of basic capture code for raspberry pi, 2018. URL `https://groups.google.com/g/flir-lepton/c/y7333qnWI9M`. 2.2.3, 4.5

[35] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91 (1):112–126, 2003. 5.1

[36] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. 5.1

[37] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 85–95. Association for Computing Machinery, 2002. 5.1

[38] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 29–39, 1986. 5.1

[39] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A Lee. Toward a Lingua Franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):1–27, 2021. 5.1

[40] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005. 5.1

[41] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, 1983. 5.1

[42] Michael McKerns and Michael Aivazis. pathos: a framework for heterogeneous computing, 2010.

URL https://uqfoundation.github.io/project/pathos. 2.5

[43] Michael M McKerns, Leo Strand, Timothy Sullivan, Andy Fang, and M Michael G Aivazis. Building a framework for predictive science. In *Proceedings of the 10th Python in Science Conference*, 2011. 2.5

[44] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming*, pages 308–333. Springer, 2014. 5.1

[45] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991. 5.1

[46] Arvind Mithal and Kim Peter Gostelow. The U-Interpreter. *Computer*, 15(02):42–49, 1982. 5.1

[47] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):1–51, 2011. 5.1

[48] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013. 5.1

[49] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. *Information and Software Technology*, 55(7):1304–1315, 2013. 5.2

[50] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 489–498. IEEE, 2007. 5.1

[51] Rishiyur S Nikhil et al. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990. 1.1, 5.1

[52] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. 4.1.9

[53] Manu Shergill. Mandeeps/su-watercam: Software for a Raspberry Pi-based environmental monitoring system, 2021. URL https://github.com/mandeeps/SU-WaterCam. 3.2

[54] Nicolas Sornin, Miguel Luis, Thomas Eirich, Thorsten Kramp, and Olivier Hersent. LoRaWAN Specification. *LoRa alliance*, 1:16, 2015. 3.2

[55] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures*, 44:143–165, 2015. 1.1

[56] Tommaso Venturini, Mathieu Jacomy, and Pablo Jensen. What do we see when we look at networks: Visual network analysis, relational ambiguity, and force-directed layouts. *Big Data & Society*, 8(1), 2021. 4.8.1

[57] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. 5.1

[58] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, 2004. 5.1

[59] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer networks*, 52(12):2292–2330, 2008. 5.1

[60] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia Derler. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 77–86, 2009. 5.1