Student Design Competition: Networked Computing on the Edge

One Program to Rule the Intersection

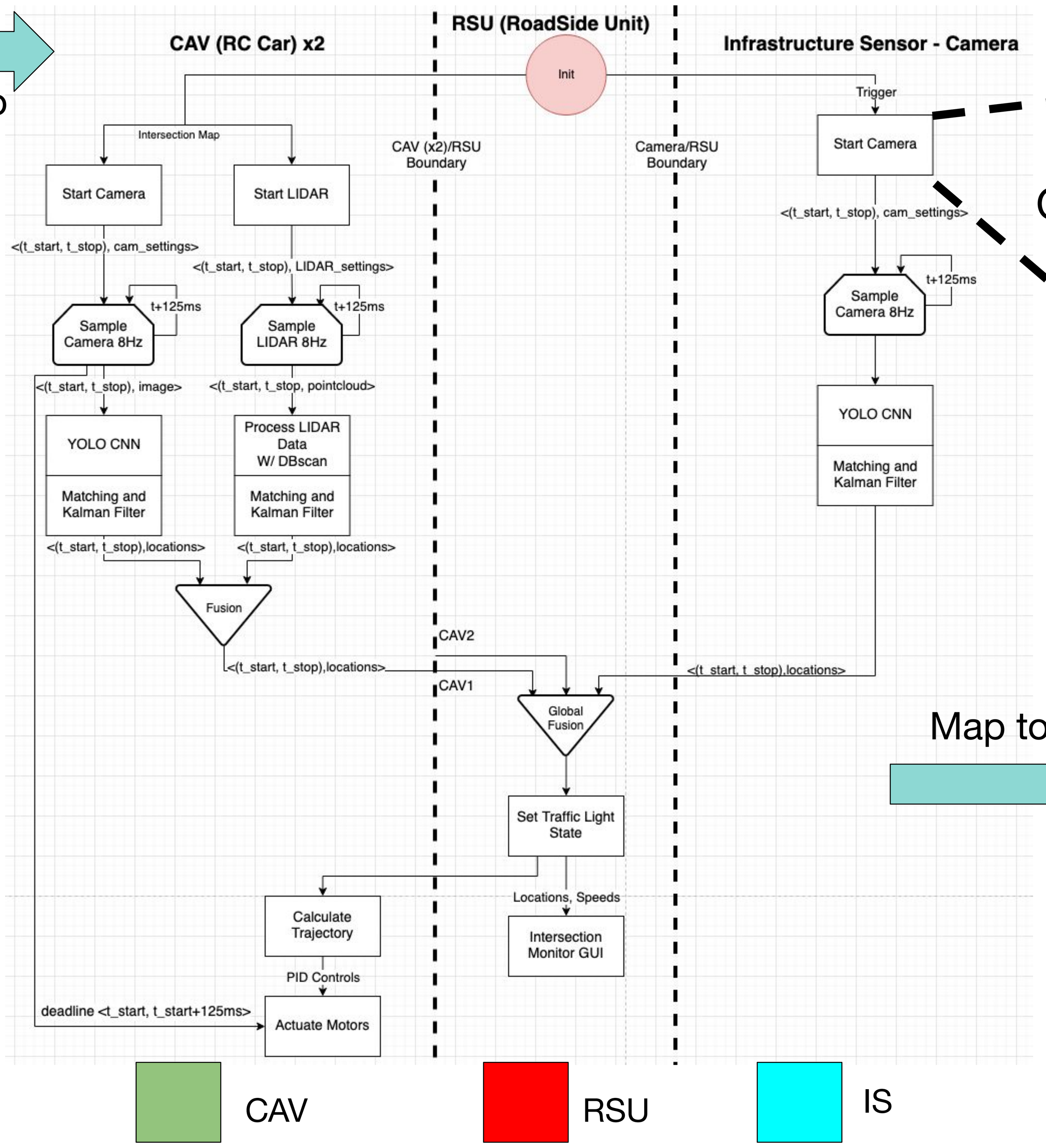
Reese Grimsley, Edward Andert, Ian McCormack, Eve Hu, Bob Iannucci

Abstract:

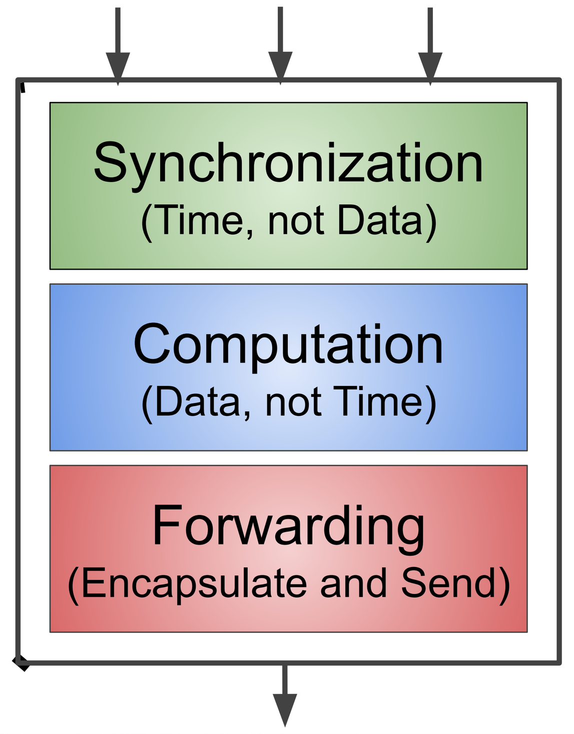
Distributed, time-sensitive applications are challenging to design, develop, and test. A signal-free intersection would have separate programs for traffic controllers, vehicles, and roadside sensors with their own independent interfaces. We designed TickTalk (TT) Python to enable systems-level programming and alleviate these challenges by abstracting communication and time-sensitive behavior to improve consistency and reduce the developer's workload. We implemented these abstractions on a smart intersection application for 1/10th scale autonomous vehicles. Our abstractions made the application easier to manage while increasing performance with respect to the sense-to-actuate latency from 127ms to 85ms, at the reasonable overhead cost of 5ms latency across the application's critical path.


```
@GRAPHify
def SmartIntersection(init_trigger, incoming_vehicle):
    map = SLAM(init_trigger)
    with TTClock.root() as ROOT: #Root clock at 1µs precision per tick:
        with TTPlanB(handler_slam_brakes, incoming_vehicle):
            with TTDeadline(ROOT, 125000): #Set deadline to force Plan B in case of failure
                with TTClock('child', ROOT, 1000, 0) as child_clock: #sample every 1000µs
                    with TTClock('LIDAR_clock', child_clock, 125, 0) as LIDAR_clock:
                        LIDAR = sampleLIDAR(TTTime(LIDAR_clock, 0, 100))
                        loc_LIDAR = localize(LIDAR, map)

                    with TTClock('CAM_Clock', child_clock, 100, 0) as CAM_Clock:
                        image = sampleCamera(TTTime(CAM_Clock, 0, 100))
                        loc_vehicles = YOLO_CNN(image) #object detection

                    #use timestamps to merge streams from different source clocks
                    merged_locations = fusion(loc_LIDAR, loc_vehicles)
                    planned_route = calculate_trajectory(merged_locations, incoming_vehicle)
                    follow_trajectory(planned_route, incoming_vehicle)
```

Compile to Graph



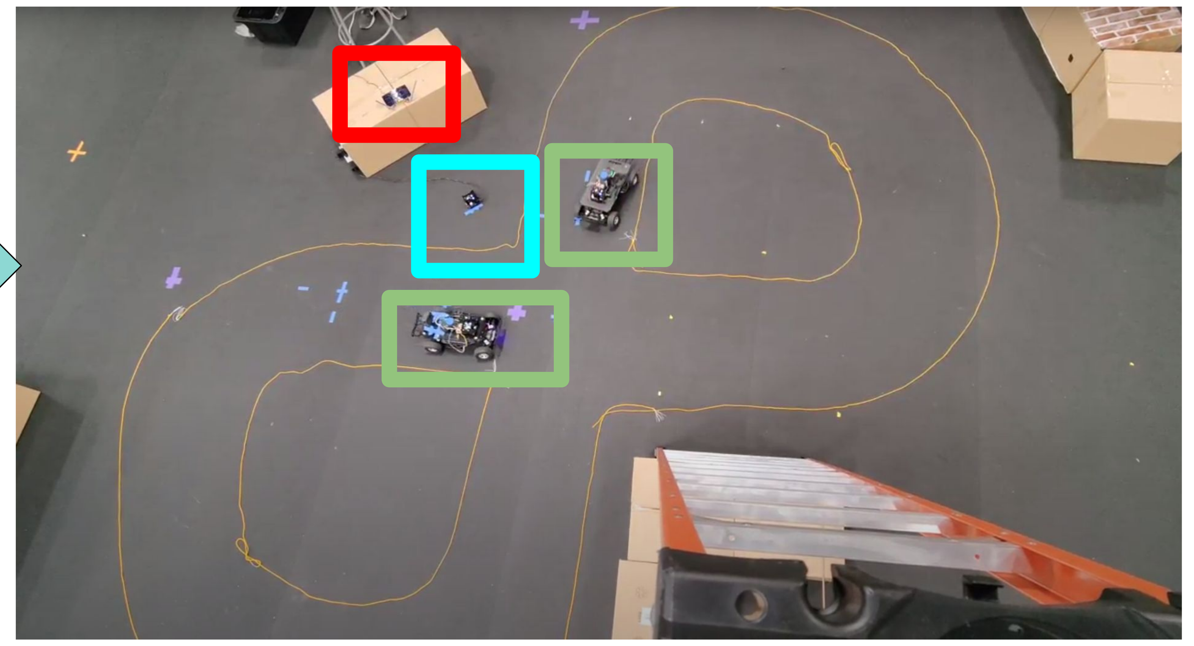
Scheduling Quantum (SQ)



SQs compose graph

- Built-in abstractions for time and communication
- Synchronize inputs using sample time
- Enforce timely behavior with deadlines on synchronization
- Sending along graph arc → implicit inter-device communication

Map to System



TTPython for Distributed, Time-sensitive Programs

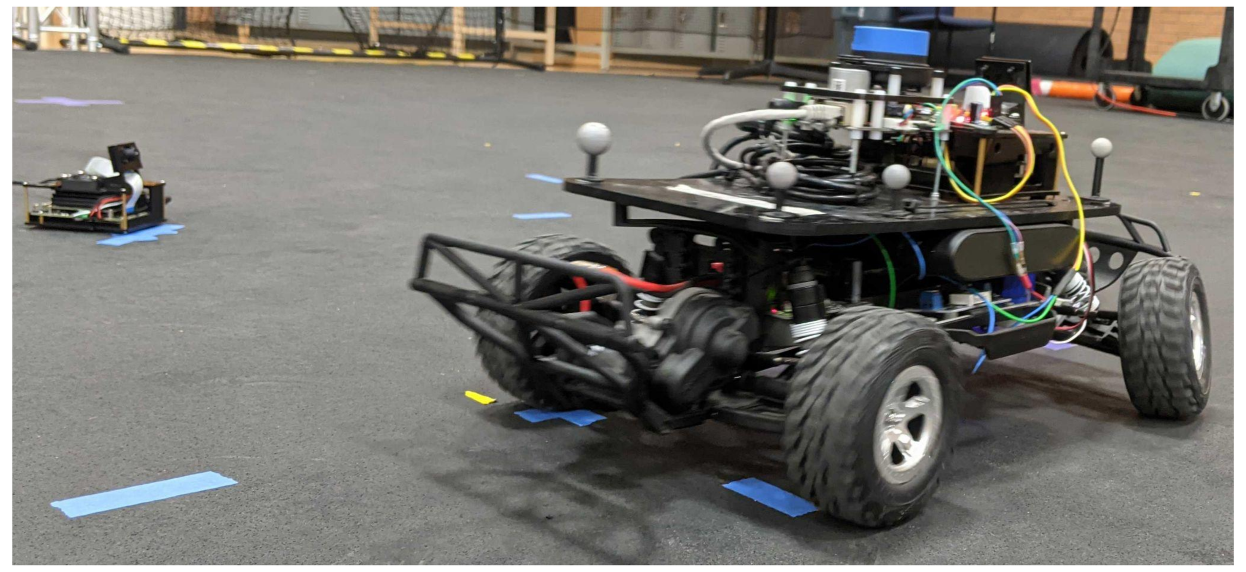
- Decompose programs into dataflow graphs
- Graphs built of "Scheduling Quanta" (SQs)
  - Wrap user-code with abstractions for synchronizing input data, making communication implicit
- Map SQs to devices in the system
  - *Intersection App*: 1/10th scale Autonomous Vehicles (CAVs), instrustructure sensor (IS), and Roadside Unit (RSU)

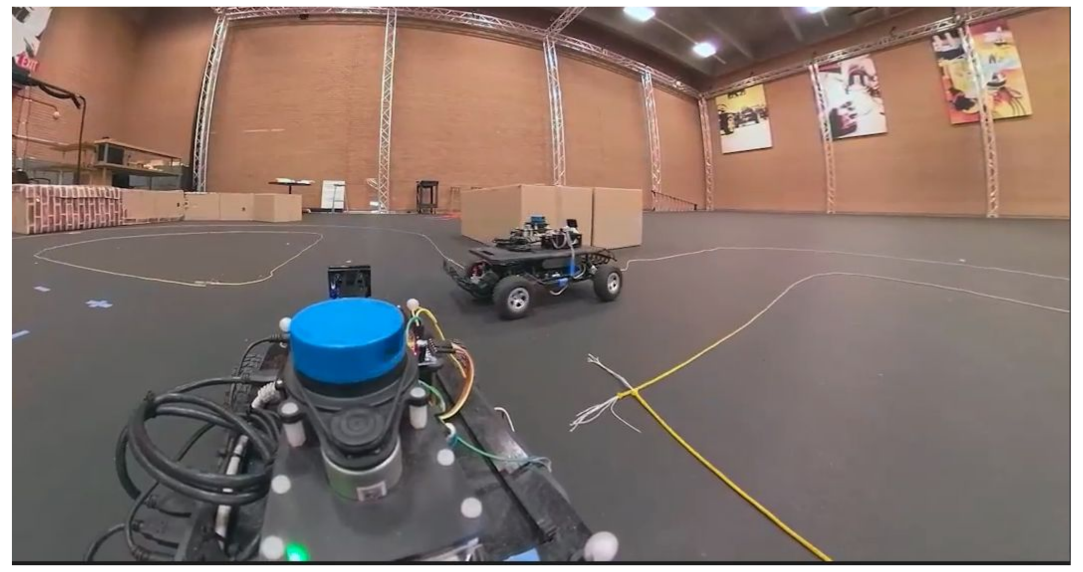
Signal-Free Intersection

- 1/10th scale vehicles with Nvidia Jetsons
  - LIDAR for SLaM localization
  - YOLOv4 Darknet CNN on images for object detection
- Roadside Unit plans trajectory through intersection to improve efficiency, throughput

Development Challenges:

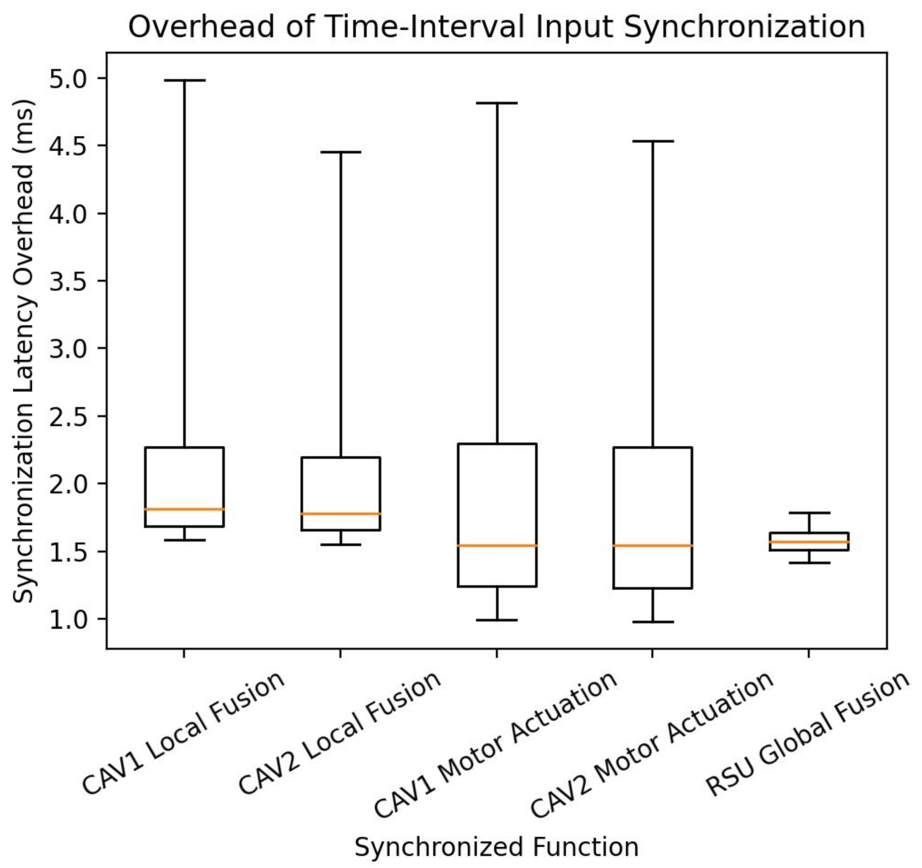
- Explicit communication in user code
- Joining sampled data from different devices with loosely-sync'd clocks
- Ensuring timely response in spite of network, sensor failure





Results

- TTPython helped discover subtle bugs
  - Improve application stability for longer tests (>5min runtime)
- Improve sense-to-actuate latency
  - Average 127 ms before, **85 ms** after
  - Due to TTPython's interdevice, interprocess communication optimizations
- Actuation deadlines hit 0.7% of the time
- 1.5-2ms overhead of input synchronization
  - Mean total overhead 5ms along critical path



Future Work

- Extend to other distributed, time-sensitive applications
  - User studies on TTPython syntax, semantics
- Dynamic mapping based on heuristics
  - Optimize metrics like latency, power-consumption
- Theoretical model for "time-governed" dataflow
- Build a community!

Conclusion

- TTPython systems-level programming for distributed, time-sensitive applications
- Built-in abstractions for time, communication
- Improve Smart-Intersection application critical path latency from 127 to 85 ms

TTPython Resources

Code: https://bitbucket.org/ccsg-res/ticktalkpython/src/master/

Docs: http://ccsg.ece.cmu.edu/ttpython/index.html

Contact:: ticktalk-python@lists.andrew.cmu.edu

Carnegie Mellon University

Arizona State University

University of Wisconsin Eau Claire

NSF Grant: TickTalk
Award ID#:
CNS-1646235, 1645578