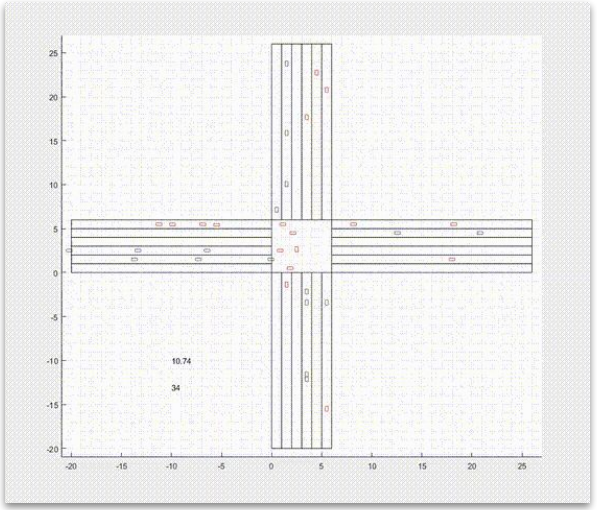
One Program to Rule the Intersection

Simplifying Development of
Distributed, Time-Sensitive Applications

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

Smart Intersections

- Light-free traffic control
  - Individualized routes, higher efficiency
- Distributed, time-sensitive application
- Precise timing requirements
  - Several ms of error yields catastrophe

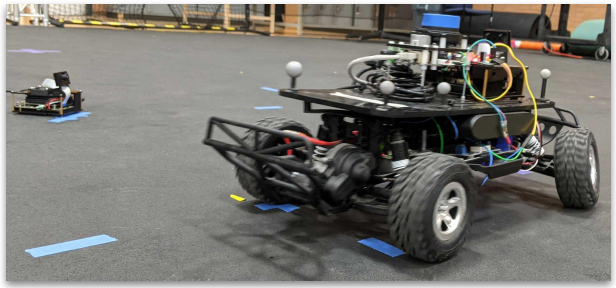


Source: 'Rush Hour' by Black Sheep Films

Source: https://safespeedllc.com/

1/10th CAV Smart Intersection Application

- Figure-8 intersection with signal-free traffic control
  - 2 Cars (CAVs) with LIDAR and cameras for SLaM, object detection
  - Roadside Unit (RSU) plans trajectories
- Development challenges
  - Timing and deadlines
  - Synchronizing sampled input streams
  - Fault tolerance
  - Explicit communication, retransmission



Design Principles

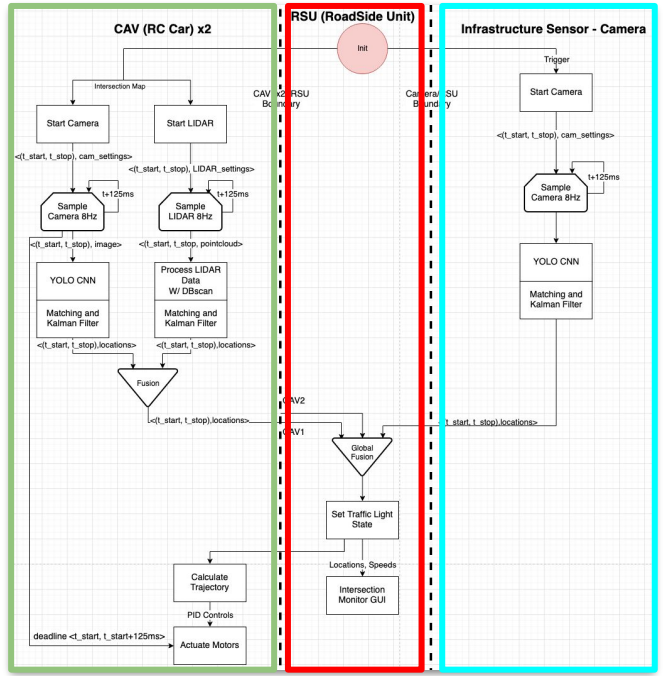
- Compatibility
  - TTPython
- Simplify time management at user level
  - Synchronization, deadline checking
- Failure handling/recovery
  - Plan B
- Abstract over communication
  - Generic network interface

TTPython

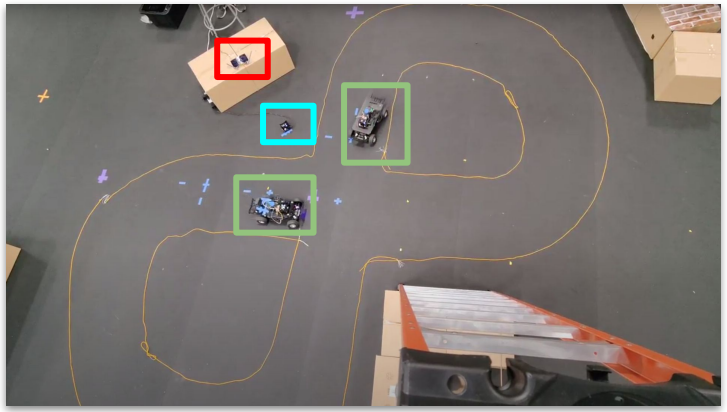
Systems-Level Programming for Distributed, Time-Sensitive Systems

```python
@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



Map to System



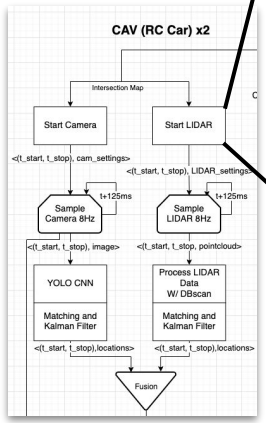
CAV

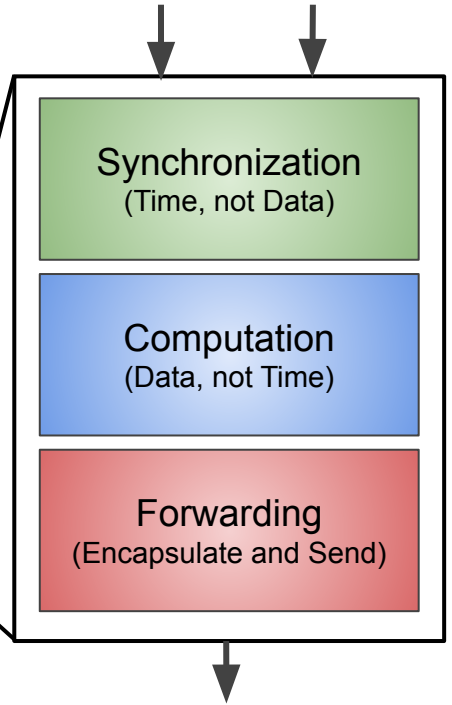
RSU

IS

Scheduling Quantum (SQ)

- Building block of dataflow graph
  - Abstractions help shift developer focus to application specifics
- Synchronize inputs
- Runs to completion once enabled
- Arcs between SQs represent implicit communication



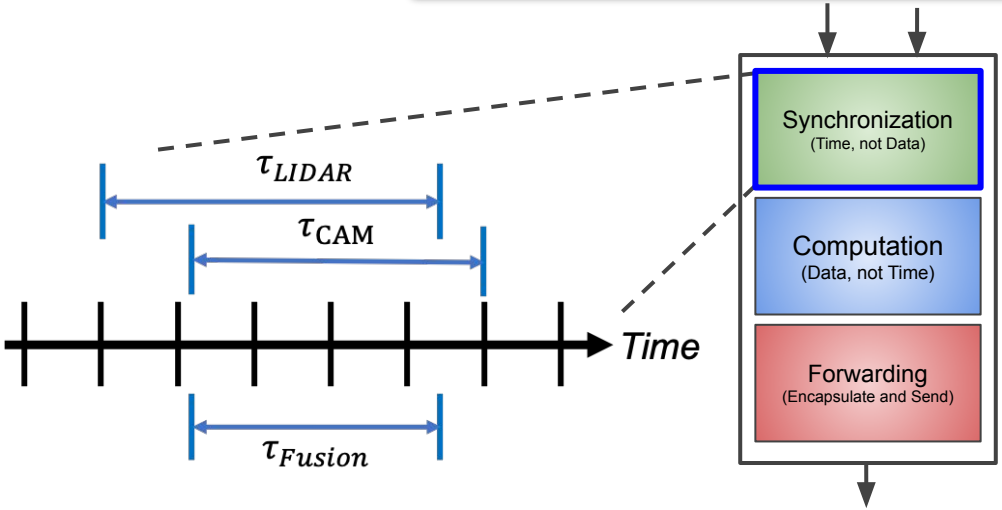


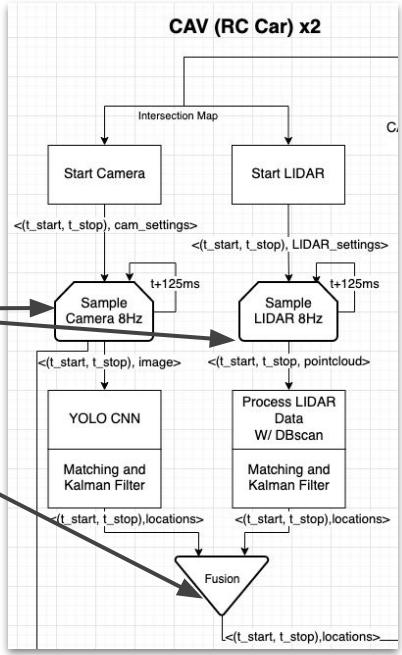
Merging Sampled Data Streams

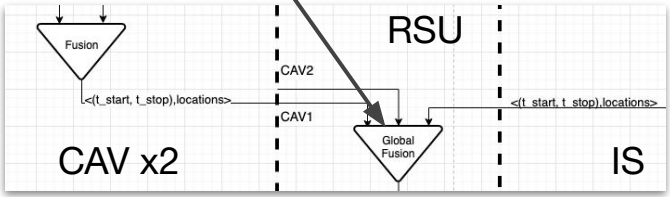
- Synchronize sampled *data* using time
  - Asynchronous devices → frequency, phase errors
  - Overlapping interval of data validity

```
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)
```

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



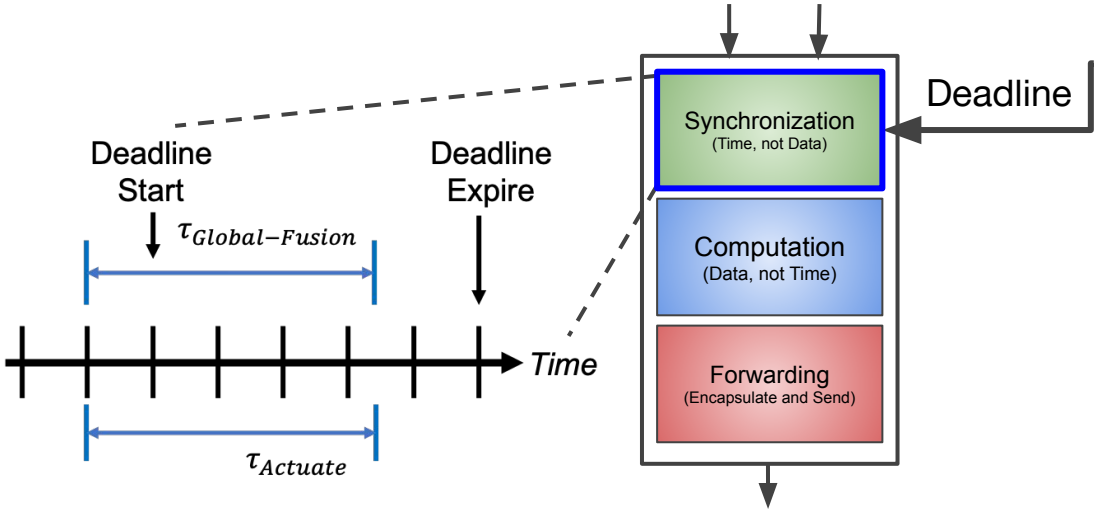


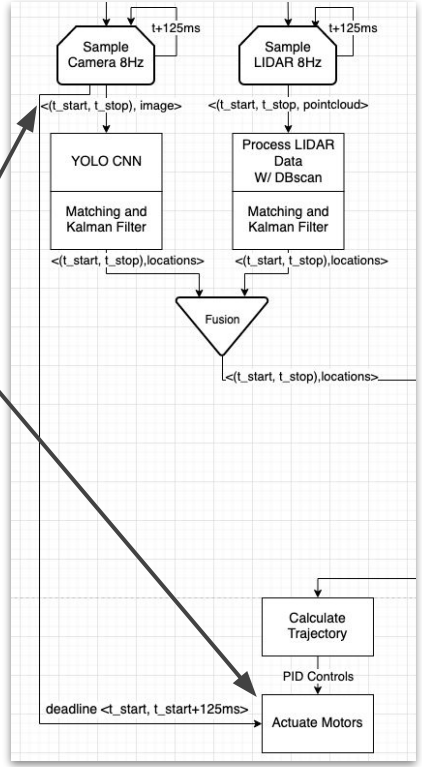


Tolerate Faults with “Plan B”

- Failures happen → support alternative action
- Enforce timely action with deadlines
  - Shortcut synchronization
  - Execute “Plan B”, e.g. apply brakes

```
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
```

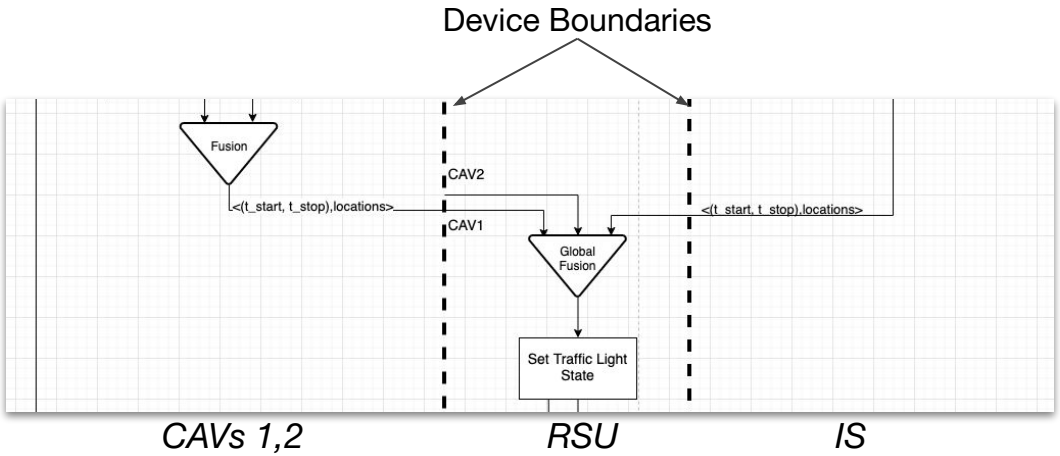


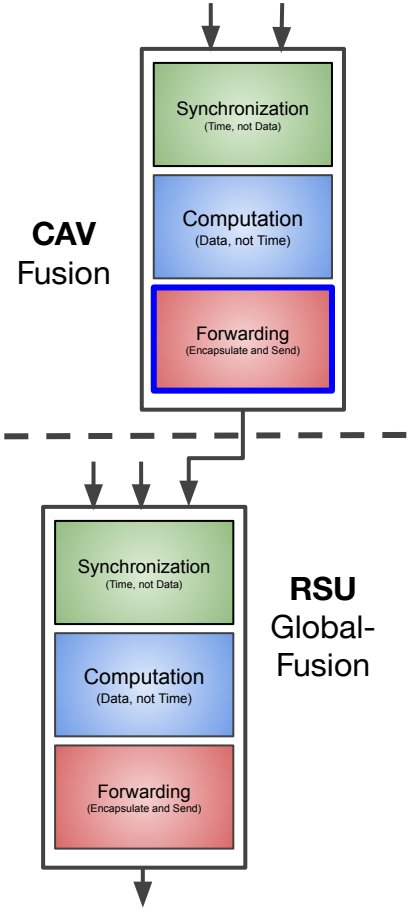


Implicit Communication

Graph arc → potential communication link

- *i.e.*, subsequent SQs mapped to different devices

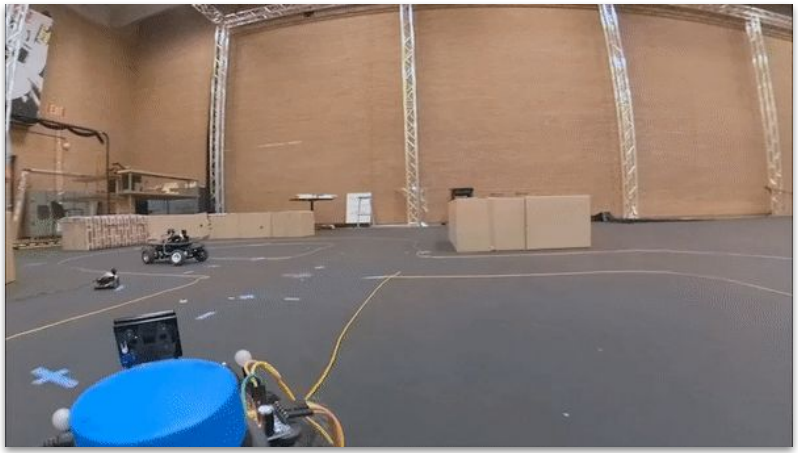




The Improved Intersection

Simplify development by abstracting time, communication

- Focus less on distributed system, temporal issues
  - SQs handle timing, deadlines, synchronization
  - Graph encodes communication links implicitly
- Exposed subtle application bugs



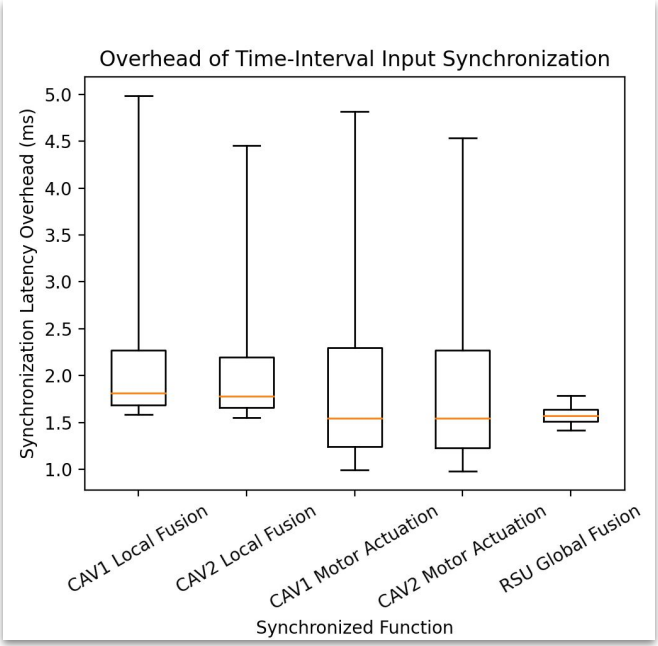
Quantitative Improvements

Round Trip Latency
127 ms to 85ms
-33%

Actuation Deadlines Hit
0.7%

Median Function Sync Latency
1.5-2ms

Mean Overhead on Critical Path
5ms



N=3000; 6.5 minutes of continuous testing

Future Work

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

Conclusion

- Distributed, time-sensitive applications are challenging
- TTPython framework for system-level programming
  - “Scheduling Quantum” (SQ) abstraction
    - Simplify communication and time-sensitive behavior
- Improved smart-intersection development process
  - Increased performance
    - End-to-end latency reduced from **127 ms to 85 ms**
  - Reasonable overhead
    - 2 ms latency for input synchronization, 5ms along critical path