Timebay: an Ad-Hoc Distributed System For Student Vehicle Testing

Andrew Ealovega

***Abstract*—The University of Michigan Dearborn features many student vehicle teams that rely on test data to inform their innovations. Currently, each team has their own dedicated systems for testing, of varying quality and price. This paper introduces Timebay, a system that ergonomically provides testing infrastructure that can be shared across teams. Timebay is a network of sensor nodes that provide both online timing data for optimisation, as well as a reliable network connection to vehicles in motion for debugging and other uses. This is achieved by exploiting the observation that these two domains together create an environment compatible with mesh networking technologies, which can be used to enhance the user experience for both use cases.**

I. Introduction

THE University of Michigan Dearborn (UMD) plays host to many competitive student organisations. These organisations challenge students to form cross-functional teams with their peers to approach a common task (such as rocketry, robotics, etc.), before pitting their solution against those of other schools across the country. Two of these teams, UMD-Racing and the Intelligent Systems Club, will be the focus of this paper.

UMD-Racing[1] is UMD's Formula SAE Combustion team. Their competition has them build, test, and race a formula car over the span of a single school year. As part of their program, UMD-Racing performs extensive testing days to evaluate their work, and practice for competition. These testing days also serve another very important purpose outside of engineering however: driver training. FSAE competitions nearly always feature a kinetic component, where teams actually race their vehicles against each other in various formats. Because these vehicles can be rather challenging to drive, drivers must spend many hours in the vehicle to get used to its handling. The primary measurement of driver skill (although not the sole measurement) is of course lap times, which will become relevant shortly.

The Intelligent Systems Club[2] (ISC) is UMD's general competitive robotics team. While ISC has competed in many kinds of events in the past, they are currently branding themselves as the schools autonomous racing team. Competing in the fledgling Autonomous Karting Series, ISC's primary project is the creation of a fully autonomous go-kart. Unlike UMD-Racing, ISC's vehicle does not feature a driver whatsoever, instead featuring a powerful, and networked, onboard computer. Like UMD-Racing, ISC performs testing days to evaluate performance. Once again, lap times are a useful metric of overall system performance. A challenge unique to ISC has been access to the onboard computer, since it must be headless. The solution to this classically has been to bring a home router to the testing site, as well as batteries to power it.

Recently, both teams have run into issues with their testing. First, UMD-Racing currently does not have any sort of timing system, so driver performances are ranked using a manual stopwatch. This is obviously a rather course metric, which also lacks any sort of contextual data about where on course one driver may be performing better than another. Second, ISC lacks any sort of infrastructure for testing their new Kart. All of the prior stated timing issues also apply here, just with evaluating a model rather than a driver. In addition, the team has been aiming to make their systems more ergonomic compared to past projects, networking included. While the past router solution has worked, carrying around a full size router just for a single connection is rather wasteful. In addition, it suffers if the testing area gets too large for the single router to cover, which is more likely this year with the kart than with less mobile robots of the past.

I propose that both of these issues can be solved together, using a system dubbed *Timebay* (a combination of the metric time, as well as highbay, the room on campus where the teams meet). To design Timebay, three primary design goals were identified from the prior requirements:

1) Provide granular online lap time data
2) Provide wide ranging network access to the vehicle
3) Be reasonably ergonomic, such that those with little knowledge of IT can still operate the system

Before we can begin to attempt to solve these goals together, we first need to investigate existing systems that solve them separately. To do this, we have identified a representative solution for each goal:

- Goal 1: Sector timing systems
- Goal 2: Mesh networks

II. Existing Solutions

A. Sector Timing

In motor sports, a sector refers to a section of the full circuit. Sector times are thus simply the amount of time it took a driver to pass though a sector's bounds. This is similar and somewhat interchangeable with the term time splits or just 'splits'. Such a timing system is often complemented by time deltas, which are the differences in time taken between two laps in the same sector. For example, if a driver discovers a better way to enter

[1]https://umdearbornfsae.weebly.com
[2]http://www.iscumd.com

a turn, their sector time will go down, and their time delta will report the empirical amount of time saved.

These systems are often used to compare two drivers performances, as well as analyse one's own performance while training. The primary advantage of using a sector timing system over a full course system is the introduction of context to the full lap time, since one can view where time was gained or lost around the course.

Most sector timing systems are implemented using a series of sensors around the course, with some sort of receiver on the car itself. As the car passes a sensor, its time is logged on the car, or alternatively some sort of central computer. An example of such a system is the MoTec BTX and BT2, which is the system UMD-Racing would use without Timebay [1]. These systems are quite expensive, as seen by the (at the time of writing) $790.00 price *per transmitter* in the MoTec system.

Upon interviewing UMD-Racing, they explained that the primary use of a sector timing for their student team would be for online information. This is because they already have advanced sensors for tracking position, but this data must be processed over a period of hours following practice. While training drivers, they could utilise the extra information given by a sector timing system to more scientifically iterate improvements. This makes a system like the above MoTec a difficult proposition, since it's a system designed for professional racing with many features UMD-Racing would not use. For UMD-Racing, timing data alone would suffice.

B. Mesh Networking

Both teams operate in regions without existing network infrastructure. Thus, when supplying network access to the vehicle, we must utilise ad-hoc networking. One popular approach to ad-hoc networking is mesh networking. Mesh networks, sometimes referred to as wireless mesh networks (WMN), are a network topology where the network is both managed and created by some set of stations. They are highly dynamic, and need to handle network degradation or complete node loss gracefully [2]. Relevant to this work are mesh networks created using WiFi, for which there are two prominent implementations: BATMAN-adv and 802.11s.

BATMAN-adv: BATMAN-adv (better approach to mobile ad-hoc networking advanced) is a routing protocol for multi-hop ad-hoc networks [3]. It operates on OSI layer 2, and is implemented in an in-tree Linux kernel module, which makes it available on any reasonably modern Linux system. BATMAN-adv is rather interesting in that it can run over both wired and wireless networks. To do this, it operates over Linux network devices, ultimately providing a network device (bat0) as its API. This allows BATMAN-adv to be used over rather heterogeneous networks. Of note is that this means BATMAN-adv can be used to allow for multi-hop routing at layer 2 in 802.11 IBSS (ad-hoc) networks, which are normally single hop only.

802.11s: 802.11s is the mesh networking protocol implemented into the 802.11 standard itself [4]. Being an 802.11 extension itself, 802.11s operates on layer 2. This does however mean that, unlike BATMAN-adv, 802.11s only works with devices both running WiFi and with drivers that support mesh mode operation. Linux does have support for 802.11s in its native driver stack, however only a few chip-set drivers actually implement this interface [5]. Unlike BATMAN-adv, 802.11s does have the nice advantage of being usable with non-Linux devices, making it better at handling device heterogeneity.

Little existing works investigate the use of these protocols in vehicular mobility. [6] Found BATMAN was ill suited to robotic teleoperation with its default settings, but could work spottily with some adjustments. In [7], the authors find BATMAN-adv to be more capable of handling mobile scenarios then rival protocols, given a predictive extension. In [8], the authors investigate 802.11s under high speeds. They found applications resistant to jitter could run over the connection, although jitter sensitive connections like voice and video could not.

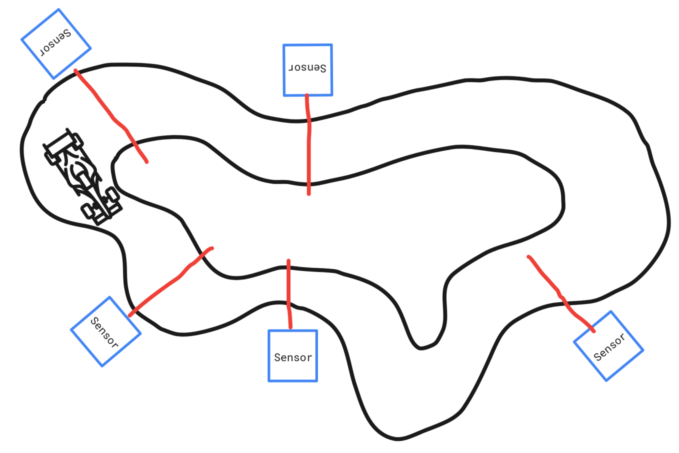
For ISC's use case, mobile connections to the vehicle would be highly desirable, but not necessarily required. This is because there are still many use cases where having the vehicle parked somewhere on the circuit with network access would be valuable.

III. Design

The design of Timebay is based around the key observation that goals 1, when implemented using a sector timing system, and 2, when implemented as a mesh network, lead to a physically similar topology (visualised in fig. 1). This naturally occurs because both systems want to be close to the car. The sector system does this for contextual data, and the mesh network does this for coverage. This makes these two solutions naturally complementary. As the car goes around the circuit, it will pass by sensors to read its sector time. If we also add networking to these sensors, then the car can also move its connection to that local sensor node. Further, these sensor nodes will be spread somewhat equally across the track. Given these nodes form a mesh network, the mesh will be able to form around obstacles that would have otherwise blocked a directional signal from a single station (such as the home router ISC uses).

This sensor-mesh approach also conveniently handles part of our third goal, ergonomics. Mesh networks are, by definition, self organising. This means that they avoid many of the intricacies of setting up an AP for an end user since they can just be pre-configured by the creator. These mesh networks will also be more resilient to interference, since they can naturally route around something like a locally strong EMF. This should assist in abstracting away the complexities of networking from the user.

There is an issue with using a mesh network approach however. For either use case, an end user cannot get information out of the the network without also using a mesh compatible machine. This goes against our ergonomic goal. To solve this, we introduce a gateway node, which is simply a node that allows for access to the mesh network without participating in the mesh itself. End users will attach their computers to this node, allowing the mesh network to be abstracted away. To



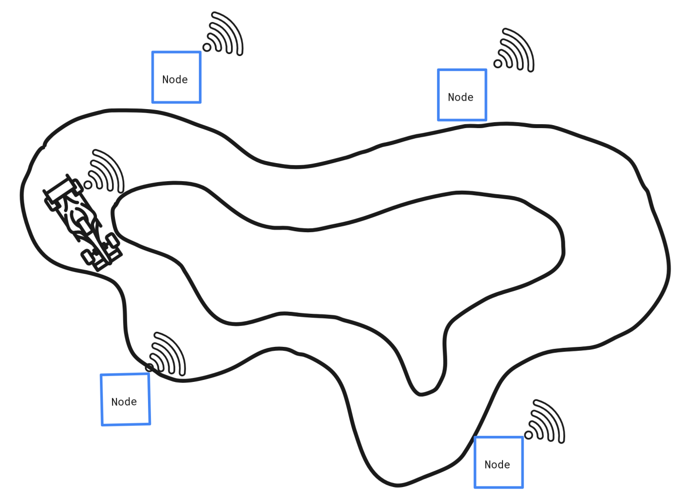


Fig. 1. A visualisation of the physical similarities between a mesh network and sector timing system

aid in debugging the mesh network from outside, a GUI will be developed that displays sensor node connection status.

A. Components

In summary, Timebay is comprised of:

Sensor Nodes - Nodes that handle both detection of a passing vehicle and organisation of the mesh network. Timebay should support 1-many sensor nodes in a network, where each sensor node is allocated a unique identifier. These nodes are fully wireless and battery powered to ease setup.

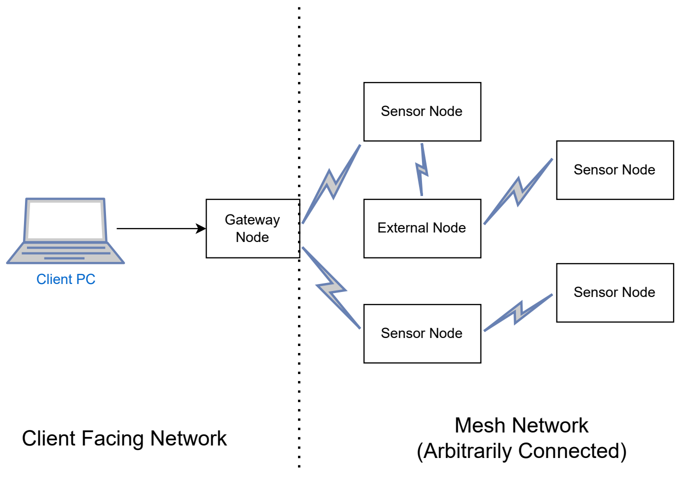


Fig. 2. High level network topology

Gateway Node - A node that serves as the single entry point to the mesh, and Timebay as a whole. This node should have two network interfaces. One participates in the mesh, and the other allows for access to the mesh (via NAT, bridging or otherwise). Any centralised components of Timebay should also run on this server, such as network management utilities or databases.

External Nodes - Guests that participate in the mesh, but are not sensor nodes. These nodes use the mesh simply for its network access to the gateway or other external nodes. An example of an external node would be ISC's onboard PC.

Clients - End user computers that have no mesh compatibility. These are off the shelf laptops that may be running one of many OSes. Clients should connect to the external network interface on the gateway node to access all network resources. A cross-platform GUI application will be provided that will handle access to all of Timebay's timing functions.

IV. Implementation

A. Hardware

Nodes - Our node implementation uses the Libre Compute Le Potato single board computer. This is a Raspi3 like board that has an 800Mhz processor, 2Gb RAM, and 1/100 Ethernet. To allow for wireless connectivity, we use ALFA AWUS036ACM (mt7612u chipset) wireless adapters.

Sensor Nodes - Vehicle detection will be preformed using a TF-Luna laser distance sensor. This sensor will be used like a line break sensor, detecting the vehicle when an object is detected some threshold closer than the zero point.

Gateway Node - Clients will connect to the gateway node using it's Ethernet port.

B. Network

The first key decision required for Timebays network implementation is a choice of physical layer protocol. Three main protocols were considered during this process.

- **WiFi** - High bandwidth commodity standard
- **Bluetooth** - Low bandwidth, low power

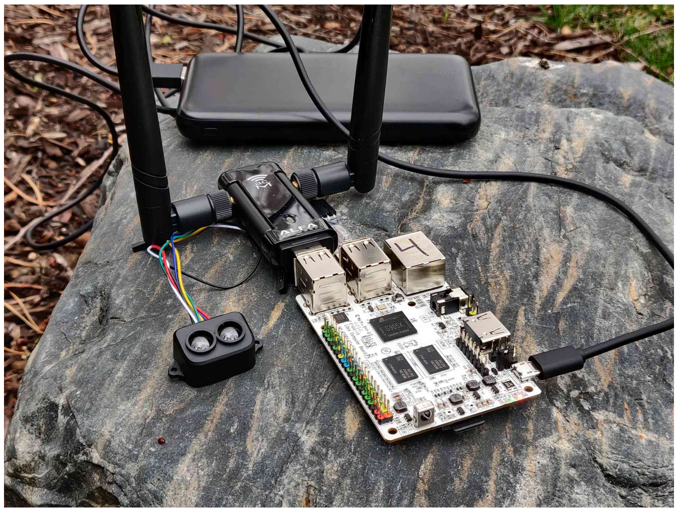


Fig. 3. A sensor node

- **Zigbee** - Lowest bandwidth, commonly used in mesh networks

Overall, Zigbee is likely the most fitting protocol, since it allows for using very low power devices for sensor nodes, and natively supports mesh networks. For Timebays use case however, Zigbee's middling 250Kbps bandwidth is simply too slow for some of the more intense activities ISC may perform [9]. For example, creating an FPV video setup would require bandwidth well into the tens of megabits, which only WiFi supplies. For this reason, WiFi was chosen as the wireless protocol.

For our layer 2 mesh protocol, Timebay implements both a BATMAN-adv and 802.11s option. This is done due to the lack of conclusive evidence supporting either protocol as being definitively superior in the context of mobility. Additionally, little documentation exists regarding the current state of each project, or their associated hardware quirks.

All nodes get dynamic IPs in the 192.168.0.0/24 subnet, allocated by a DHCP server on the gateway node. The gateway node itself has the static IP 192.168.0.1, to allow for connections to centralized resources. DHCP was chosen over static IPs to allow for scaling the number of external nodes in mesh without requiring the user to maintain a record of allocated addresses.

To allow for clients to connect to the mesh network, the gateway node creates a Linux bridge interface between its Ethernet port and its mesh interface. The gateway node is only accessed via the IP address on the bridge, so clients connecting to the mesh can access any other mesh node as if they themselves were on the mesh. This also allows for clients to receive DHCP addresses, as DHCP will flow over the bridge.

C. BATMAN-adv Configuration

BATMAN-adv is by default not very mobile friendly, as discussed in [6]. Borrowing from that work, we will configure BATMAN to send OGM every 50ms. The goal of this is to minimize the time it takes for the mesh to reconfigure routes, critical to ensure dead nodes are routed around and mobile external nodes quickly change mesh links. To aid in this, OGM aggregation is also disabled, as recommended by the BATMAN-adv docs [10]. Both of these configurations will likely lower throughput, but should hopefully improve persistence in exchange. Following the results of [11], we will be using BATMAN IV over V.

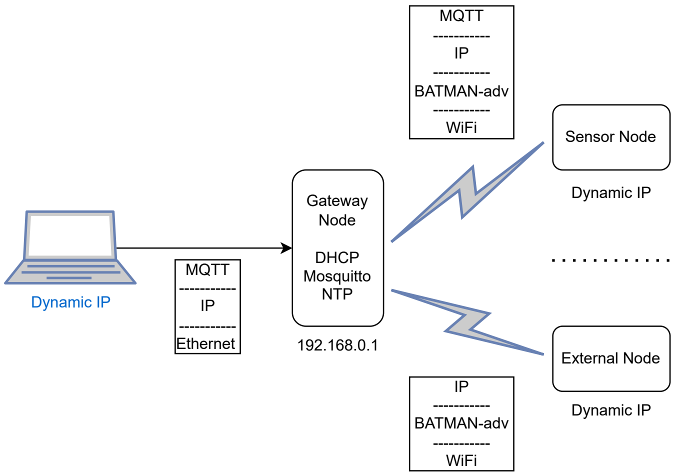


Fig. 4. Timebay network implementation

D. Sensor Node Communication

All data related to the timing features of Timebay are broadcast across the network using MQTT. MQTT was chosen in this case not for its low bandwidth requirements (although this is nice), but rather its convenient pub-sub architecture which naturally mixes well with our highly dynamic network topology. The MQTT broker for Timebay is hosted on the gateway node.

The following topics are used:

/connect - Published to by nodes when they connect or reconnect to the broker. Nodes are allowed to publish to this topic when already connected, as a sort of heartbeat.

/disconnect - Published to by nodes when they disconnect from the broker, via LWT.

/zero - Causes all sensor nodes to zero their sensors.

/sensors/detection - Published to when a node detects a passing vehicle.

E. Time

As the name would suggest, time is rather important in Timebay. When sensor nodes send their data to the client to create sector times, we need to ensure that the client has an accurate estimation of the time that detection actually occurs. There are two ways to approach this problem:

1) Timestamp when we receive sensor detections on the client
2) Timestamp before we send the detection over MQTT, on the sensor node

When evaluating this issue, it's important to remember that what we need to combat here is not the offset between the client and sensor nodes, but rather the *offset* between each of the sensor nodes with respect to other sensor nodes. This is because lap times on the client are stored as absolute time values of the start and end of the sector. If one node's time has error from either approach, and that error is different than the error of any other node (what I call offset), the lap time will be inaccurate. From this view, it should be obvious that approach 1 is not viable. This is because every node on the network will naturally have different network latencies to the client, since synchronising latencies is not the goal of a mesh network. For approach 2, network latencies are not a direct issue. Instead, what adds offset is clock desyncronisation (or rather, clock offset as the term is normally used). Because the SBC's used for sensor nodes do not have a RTC, this necessitates the use of a clock synchronisation solution.

Clock synchronisation in WSN is a well studied area [12] [13] [14] [15]. While these approaches do show promise, unfortunately few have open implementations, especially on Linux. Because we lack the resources to implement one of these protocols ourselves, we decide to simply use NTP, the traditional time synchronisation protocol on the Internet. As discussed in [12], an issue with NTP in mesh networks is the potential for asymmetrical latencies in RTTs. To help curb this issue, we chose Chrony as our NTP implementation. Chrony uniquely supports estimating delay asymmetry in its connections, going beyond the requirements of NTP itself [16].

Clock synchronisation in Timebay is implemented by synchronising all nodes to the gateway node, since that will be the node that is always present. Because we only care about reducing the offsets between sensor nodes, it does not matter that the gateway node itself may be desyncronised. Accordingly, Chrony is configured to use its local mode, which will serve NTP using the local clock as if it was synchronised. We also declare the servers stratum as 1, since we want all nodes to see the gateway as authoritative. Each sensor node runs Chrony configured to only access the gateway's server, forcing the clock to step if error is greater than 1 second. This is done in hope of forcing fast convergence on system setup, since all nodes will be greatly desynced on boot due to the lack of a RTC.

F. Sensor Node Detection

The sensor node vehicle detection code is implemented as a Rust executable. Using the Tokio reactor, it polls over both an MQTT channel and the laser rangefinder concurrently, zeroing the sensor on MQTT command or sending a detection report to MQTT respectfully.

When detections have not occurred for some period of time (currently 3s), sensor nodes will send a heartbeat message on /connect. This is done not to tell if a node is disconnected, as MQTT does that with its own heartbeats, but rather to allow for late-joining clients to discover sensor nodes without requiring those nodes to have a detection.

G. Deployment

All Timebay nodes are implemented as Docker containers to help abstract over hardware heterogeneity, as well as to ease the difficulty of adding new sensor nodes. These containers are set up to boot on board restart via cronjob.

Because we need to access wireless interfaces from the container, we disable most virtualisation features of Docker. This involves running the container as privileged, using host networking, adding SYS_TIME capability (required for NTP), and mounting /dev/.

To ease updating potentially many nodes, we set up a sort of 'OTA' update system. When a node boots, we configure its Ethernet port with DHCP if a carrier is available. In its bringup script, we fetch the Timebay repo's upstream. If we have a carrier on eth0 and the remote is ahead of HEAD, then we pull the remote and rebuild the docker container. This means that to update many nodes, one simply needs to plug them all into a switch and supply power.

The general boot process for sensor nodes is:

1) Run bash script that git pulls Timebay, and rebuilds the docker container if new changes were pulled
2) Run docker container
3) Find a mesh/IBSS compatible interface by scraping iw output
4) Use iw to set the device to IBSS/Mesh mode
5) If using BATMAN-adv, setup bat0 interface
6) Start dhcpcd to get IP address
7) Start sensor node executable

H. GUI

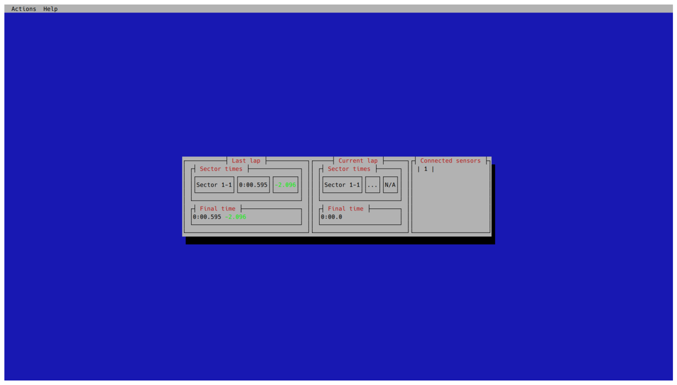
The client GUI application is also implemented in Rust, and utilises shared components with the sensor node. The GUI implements all of the actual application logic for sector timing, aggregating data collected from the MQTT broker. Sectors are defined by pairs of nodes starting at the lowest node id and pairing until the last node, which is paired with the initial node id. Each sector begins timing at the end trigger of the last sector, updating the GUI as the course is completed.

If the car skips a node, the time spent in the sector is simply rolled into the next sector, and the skipped sector is marked invalid. This transparently handles nodes disconnecting, since they will simply be skipped. The alternative would be to invalidate the sector eagerly when the node disconnects but before the car passes it, but this idea was discarded as it complicates the logic of the node coming back online. By treating disconnected nodes as if they were skipped, we optimistically give the node a chance to reconnect with the network without any sort of special logic.

Should the nodes clocks become desynced, it is possible for the GUI to detect the detection timestamps going back in time. If this occurs, the current sector is marked as invalid.

When a lap is completed, the GUI calculates the complete lap time and moves the splits to one side of the screen, and then displays the time deltas with respect to the prior lap.

Also included in the GUI is a display of all currently connected nodes, as well as a button to zero the sensors. Nodes will display as connected as soon as they send a message



```
Last lap                              Current lap                 Connected sensors
  Sector times                          Sector times              | 1 |
  Sector 1-1 | 0:00.595 | -2.096        Sector 1-1 | ... | N/A
  Final time                            Final time
  0:00.595 -2.096                       0:00.0
```

Fig. 5. Timebay's GUI connected to a single sensor node

Trial (BATMAN-adv)	Time Till complete
1	3:30
2	3:55
3	3:44
4	4:21
5	3:44

Trial (802.11s)	Time Till complete
1	3:11
2	3:20
3	3:25
4	3:20
5	3:24

Fig. 6. Timebay startup times

to /connect, and will show as disconnected 10 seconds after disconnecting, since we have configured a 10s TTL.

In an attempt to make this GUI as cross platform as possible, it is actually implemented in the terminal, making it a TUI. This avoids linking with platform specific GUI libraries, increasing the chance of the GUI remaining compatible on future, unknown OS versions.

I. Ergonomics

Timebay's third goal, ergonomics, is primarily achieved by designing the network to be as self operational as possible. For example, all mesh networking is abstracted away by the gateway node. If a node disconnects, one simply needs to wait for it to re-mesh, since the GUI will automatically handle how that addition or reduction of a node will impact lap times. If the layout must be changed, all the user must do is press the zero sensors button in the GUI and the system will be operational again. Mobile batteries are used to power the nodes as opposed to LiPos to allow for easy recharging of nodes by non electrical members.

In addition to these and many more design choices, we make a point to supply ample documentation of the system. This is important, since the student teams have very high turnover, and organisational knowledge around tools is very volatile. In particular, instructions to replicate the current implementation are provided, as well instructions for operation, troubleshooting, and development. To ensure these are not lost while the hardware remains in use, Timebay is open sourced via a GitHub repository[3].

V. Evaluation

A. Startup time

First, we will evaluate the startup time in optimal conditions. To measure this, we set up three sensor nodes in a 3m circle around the gateway node attached to a client. A timer is started on the computer, then all nodes are turned on. With the GUI open, we wait until all nodes are displayed in the connected nodes table. The results are shown in Fig. 6. Note that this includes the time for:

1) Booting Linux
2) Starting Docker
3) Connecting to mesh
4) Getting IP over DHCP
5) Connecting to MQTT

The average startup time was 3:51 for BATMAN, and 3:20 for 802.11s. Upon investigating the boot logs, it appears that approximately 2:40 of this boot time is spent booting the host OS. This is likely due to a misconfigured service. Because both protocols use the same Docker container, the consistently faster boot times on 802.11s is likely due to properties of its mesh. In particular, DHCP is a likely candidate for this lost time. As we will see later, BATMAN-adv has very large packet loss. DHCP requests addresses using a UDP broadcast. Given this best effort approach, dhcpcd's exponential back-off on failure, and the large packet loss, DHCP can take quite a while to gain an address in BATMAN-adv meshes. This was verified by Wireshark analysis, where DHCP requests could be seen leaving the node but not receiving a response. This could also explain the greater variance shown in the BATMAN-adv results.

B. Clock Synchronisation

Next, we will verify that nodes are being synchronised. This test can be seen as a validation of goal 1, since the sector timing logic works as expected, leaving only the clock synchronisation as a potential issue. This is done by placing two sensor nodes next to one another, then dropping a piece of cardboard in front of them. Because the nodes should trigger at roughly the same time, we can compare the GUI's received timestamps to find the offset of the clocks, and thus the error our timing system will see per sector. This process is begun first by rebooting the nodes to force a fresh NTP state, then collecting five samples when the nodes appear in the GUI. The results of this experiment are displayed in fig. 7. Note that stamps include only the nanoseconds component, as no readings had greater than a second of error. Offsets are rounded to the ms as our testing apparatus does not have ns accuracy.

These results show us a few things. First, our aggressive stepping of clocks until second accuracy is achieved has achieved its goal of getting the clocks reasonably synchronised within a short period of time. Second, even with very little time to converge, both protocols have an acceptable level of error.

[3]https://github.com/andyblarblar/timebay

BAT	Node 1 stamp	Node 2 stamp	Offset
1	133668199ns	158070397ns	24ms
2	568918805ns	570767970ns	2ms
3	728301891ns	746114176ns	18ms
4	503307445ns	501268735ns	2ms
5	331404650ns	335236888ns	4ms

802.11s	Node 1 stamp	Node 2 stamp	Offset
1	710317045ns	758237086ns	8ms
2	321956813ns	334979791ns	1ms
3	409371012ns	430514388ns	2ms
4	328401762ns	355385103ns	3ms
5	716304263ns	731996424ns	2ms

Fig. 7. Timebay clock synchronisation offsets

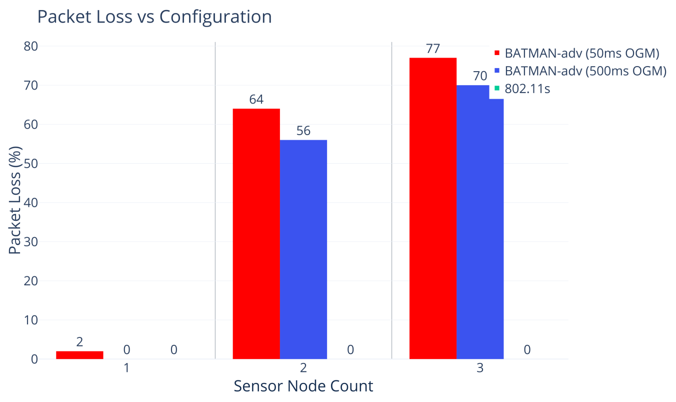


Fig. 8. Packet loss under various configurations

802.11s especially, with all measurements under 10ms. While this advantage could be from NTP having issues contending with BATMAN's packet loss, our testing apparatus has too much potential for procedural error for this to be conclusive.

C. Packet Loss

While performing other tests, many were failing to pass due to massive packet loss. To investigate this issue further, we test the packet loss under various setups to find any patterns. For all of these configurations, the client is connected to the gateway node and SSHs into it. From the gateway node, we use ubuntu's ping utility to test the packet loss to one of the sensor nodes until 150 sequences are reported. For each configuration, we repeat this test for 1, 2, and 3 connected sensor nodes. The results of these tests are shown in fig. 8.

First, we test BATMAN-adv with 50ms OGM, which was the configuration initially proposed. As shown in fig. 8, there is a massive jump in packet loss when a second node is added, with a further increase with a third. To investigate if this issue was caused by our aggressive OGM rate, we run the test again with 500ms OGMs. Once again, packet loss jumps massively when a second and then third node are added, although less so than with the 50ms OGM. To ensure this isn't a fundamental issue with our system, we finally repeat the test with 802.11s. These results are a little hard to view in the chart, since there is *literally no* packet loss, regardless of node count. This result was quite surprising, since other literature on BATMAN-adv does not document packet loss issues. The packet loss only begins once IP data begins to be sent, not when the IBSS forms. Because of this, its possible this is a misconfiguration of BATMAN-adv, although we use all default settings aside from OGM rate, which makes this unlikely.

D. Throughput In Motion

Finally, we will evaluate throughput of the mesh with a vehicle in motion. This can be seen as a validation of goal 2. Unfortunately, neither teams vehicle was available for testing. As a stand in, a collaborator's *2023 Ford Maverick XLT 2.0L EcoBoost AWD* will serve as our vehicle. To ensure fair results, we have a laptop held in the bed of the truck to prevent the body from blocking signals. This is important because both

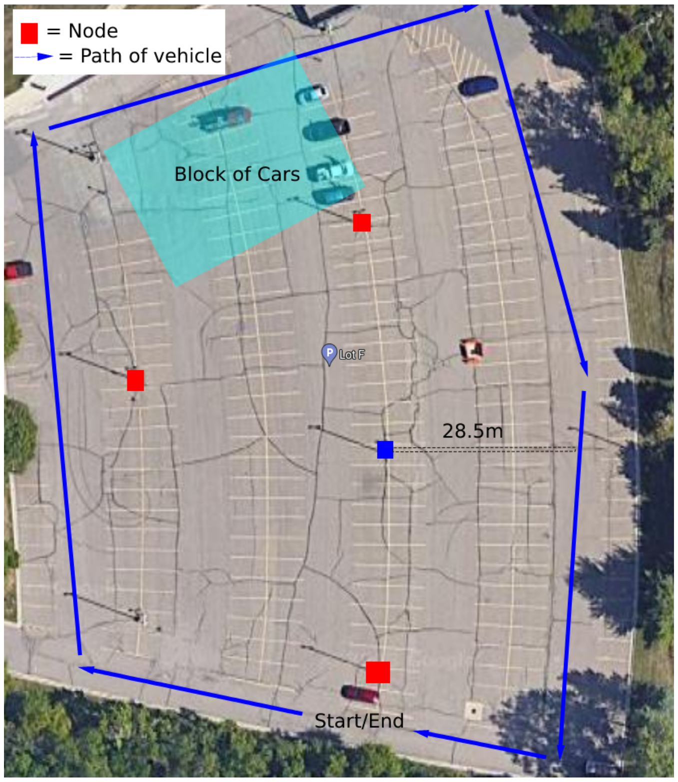


Fig. 9. Testing setup for throughput tests

student vehicles allow for unobstructed antennas. Testing was done on a parking lot on campus, with 3 sensor nodes and the gateway node spread throughout. The distribution of these nodes is displayed in fig. 9. The laptop held in the bed will be simulating the onboard PC of ISC, connected to the mesh as an external node. We will only perform these tests using 802.11s, since BATMAN-adv's packet loss was too high to perform the test with 5 total nodes. For each test, the vehicle will drive the perimeter of the lot while iperf3 runs a TCP bandwidth test of the network between a client connected to the gateway and the external node on the vehicle. Pings will run for all tests to collect aggregate latency measurements. The results of these tests can be seen in fig. 10.

As can be easily observed, Timebay has very large amounts of jitter while in motion. This was not unexpected, and tracks

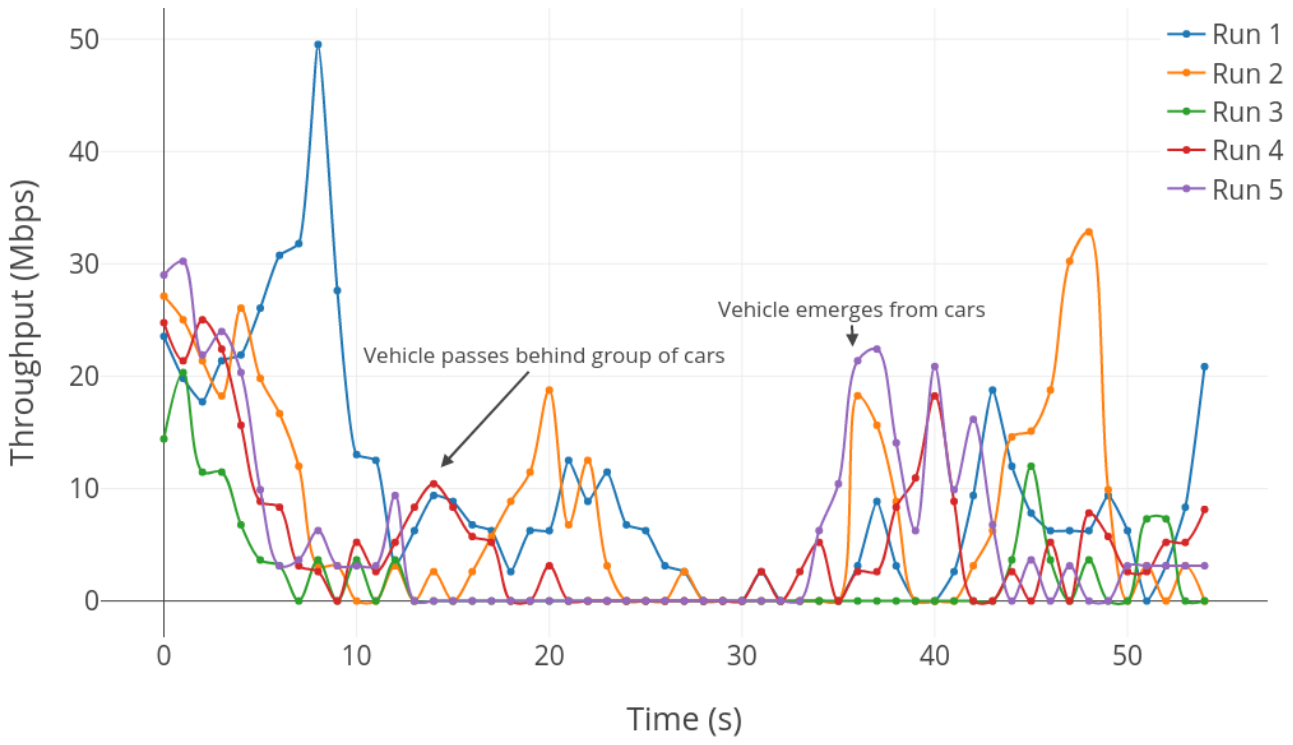


Fig. 10. Throughput of the mesh with vehicle in motion

with the results of [8]. This can be seen in the latency results, where we see a minimum RTT of 1.3ms, a max of 1800ms, an average of 68ms, and a standard deviation of 123ms. From the period of around 15s - 34s, a large drop in throughput, or even the connection, can be seen in all runs. This is due to a large block of parked cars in the lot fully obscuring the vehicle from all nodes. This gives us the opportunity to observe the mesh reforming under motion, which it does very smoothly shortly after the vehicle emerges at approx. 35s. Note that because of this, 35s onwards is the most representative of an average use case, as the vehicle is in motion and in range of at least one node. This range still shows large variance in throughput, ranging from 2Mbps to 20Mbps depending on the run. Overall, these results show that Timebay is not suitable for real time data transfer from an external node, such as for teleoperation. It is, however, capable of transferring data at reasonable speeds (faster than alternative wireless standards) in jitter durable applications. An example of this could be viewing ROS topics from the external node in Rviz, where latency is less of a concern than persistence and reasonably fast speeds.

It is worth pointing out that the above results apply to a vehicle in motion only. While stationary, Timebay can maintain consistently high speeds, sometimes even pushing up against the bandwidth of the 1/100 gateway link. This applies to both the vehicle while stationary, as well as other possible uses of external nodes, such as remote terminals around the circuit.

E. BATMAN-adv vs. 802.11s

As can be seen throughout the evaluation, the choice of BATMAN-adv vs. 802.11s significantly changes the behavior of Timebay. Of course, Timebay can only use one of these protocols at a time, so it is worthwhile defining a default.

A running theme throughout the creation of Timebay was the mesh protocols acting significantly differently given different hardware. Initially, the system used wireless adapters in the rt2800 chip-set family. While using these drivers, 802.11s was unable to bridge with the Ethernet port on the gateway, preventing the use of 802.11s in Timebay. When the project moved to adapters that used the Linux mt7x drivers for unrelated reasons, 802.11s was fully operational. This poses a significant issue for recommending a protocol, since reproductions of Timebay in the future may outright not function if chip-sets change even in vetted adapters. Unfortunately, BATMAN-adv was nearly unusable with multiple nodes in either setup due to its massive packet loss. However, it was capable of bridging even with the rt2800 adapter. Awkwardly, this leaves BATMAN-adv with a niche where it is still technically better than 802.11s, by virtue of running at all.

Overall then, 802.11s is currently far and away the better implemented protocol on Linux, so long as you have an adapter with the correct chip-set. If you don't, then BATMAN-adv can be used as a fallback, although it shouldn't be used with more than one sensor node.

VI. Conclusion

A. Limitations

1) Security: Timebay lacks any security whatsoever. This means that anyone is capable of connecting to the mesh and accessing its resources. Realistically, this primarily makes external nodes like ISC's PC vulnerable to external connections if not secured. For a more professional application, this would also make the system vulnerable to passive and active attacks on the sensor detection times over MQTT. This could be an issue if say, a formula team is practicing around competitors. For the intended use case, I do not consider this a large issue, since the onboard PC is already secured and messing with lap times is harmless. The system should be amendable to hardening many of its interfaces, such as requiring a password for MQTT and encrypting WiFi using 802.11s's SAE [5].

2) SBCs: Timebay, by its reliance on Linux, requires the use of SBCs that are likely far more powerful than needed. This reliance comes from a few sources:

1) Mesh Implimentations
2) Current sensor node implementation codebase
3) NTP

While 2 could be programmed away, 1 and 3 are far more difficult. Both mesh algorithms and clock synchronisation suffer from a lot of theoretical algorithms but a lack of implementations. Even when implementations are available, they are often for a particular embedded OS, lowering the odds of the other system also being available. Should this issue be resolved, Timebay could likely be implemented using a microcontroller like the ESP32 as the sensor node platform, although throughput would need to be re-verified.

B. Future Work

Timebay is still a very young system, and has many areas that could be significantly improved given future work.

1) 802.11s Parameters: All tests were performed using default 802.11s parameters, largely due to a lack of time for iteration. These parameters, especially those controlling the HWMP, can likely be significantly optimised for the moving vehicle scenario. 802.11s also natively supports the concept of gateway nodes, something that we do not utilise in this work.

C. WiFi

All mesh configurations currently use 2.4ghz channel 5 with HT40+ for their underlying WiFi connections. Channel 5 was chosen after a site survey of campus found it to be the least congested. Even with this choice, the 2.4ghz spectrum is very polluted from various industrial machines around campus. Future work should investigate both the use of the 5ghz spectrum as well as various channel widths to potentially increase throughput and other metrics.

1) Hardware: To improve ergonomics, Timebay could greatly benefit from standardised boxes for the nodes. This would improve setup times, lower the chance of environmental damage, and overall improve ease of use. These should be able to be constructed with 3d printers, allowing us to open source the design alongside the codebase.

D. Summary

To aid two student teams at the University of Michigan Dearborn, we analyse some shared deficiencies between the teams. This uncovered the need for both a granular timing system, and a large-scale network for vehicle connections. To solve both issues together, we combine the existing solutions of sector timing and mesh networking into one system, called Timebay. Upon evaluation, Timebay is shown to excellently fulfill its timing duties, with only a 3.2ms average error per sector using 802.11s. The mesh network is shown to provide network access to the vehicle while still and within the range of a node, but suffers from jitter in throughput and latency while in motion. Overall, Timebay should adequately address the needs of UMD-Racing, and greatly improve the networking conditions of ISC. Future work should focus on improving network conditions while the vehicle is in motion, potentially opening up new avenues of use for Timebay within ISC.

References

[1] "Motec; current range; directory." [Online]. Available: https://www.motec.com.au/laptiming-currentrange/laptiming-directory/

[2] I. Akyildiz and X. Wang, "A survey on wireless mesh networks," *IEEE Communications Magazine*, vol. 43, no. 9, pp. S23–S30, 2005.

[3] "Mesh." [Online]. Available: https://www.open-mesh.org/projects/open-mesh/wiki

[4] G. Hiertz, T. Denteneer, S. Max, R. Taori, J. Cardona, L. Berlemann, and B. Walke, "Ieee 802.11s: the wlan mesh standard," *Wireless Communications, IEEE*, vol. 17, pp. 104 – 111, 03 2010.

[5] "Linux wireless." [Online]. Available: https://wireless.wiki.kernel.org/en/developers/documentation/ieee80211/802.11s

[6] F. Zeiger, N. Krämer, M. Sauer, and K. Schilling, "Mobile robot teleoperation via wireless multihop networks-parameter tuning of protocols and real world application scenarios," in *Informatics in Control, Automation and Robotics: Selcted Papers from the International Conference on Informatics in Control, Automation and Robotics 2008*. Springer, 2009, pp. 139–152.

[7] B. Sliwa, S. Falten, and C. Wietfeld, "Performance evaluation and optimization of b.a.t.m.a.n. v routing for aerial and ground-based mobile ad-hoc networks," in *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, 2019, pp. 1–7.

[8] A. N. Mian, T. Liaqat, and A. Hameed, "Testbed analysis of 2-hop ieee 802.11s network for supporting ip services under mobility," in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, 2018, pp. 1–5.

[9] K. F. Haque, A. Abdelgawad, and K. Yelamarthi, "Comprehensive performance analysis of zigbee communication: An experimental approach with xbee s2c module," Apr 2022. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9105712/

[10] "Open-mesh " batman-adv." [Online]. Available: https://www.open-mesh.org/projects/batman-adv/wiki/Tweaking#originator-interval

[11] L. Liu, J. Liu, H. Qian, and J. Zhu, "Performance evaluation of batman-adv wireless mesh network routing algorithms," in *2018 5th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2018 4th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, 2018, pp. 122–127.

[12] T. Beke, E. Dijk, T. Ozcelebi, and R. Verhoeven, "Time synchronization in iot mesh networks," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, 2020, pp. 1–8.

[13] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 147–163, 2002.

[14] S. Ping, "Delay measurement time synchronization for wireless sensor networks," *Intel Research Berkeley Lab*, vol. 6, pp. 1–10, 2003.

[15] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 39–49.

[16] D. Mills, J. Martin, J. Burbank, and W. Kasch, “Network time protocol version 4: Protocol and algorithms specification,” Internet Requests for Comments, RFC Editor, RFC 5905, June 2010, http://www.rfc-editor.org/rfc/rfc5905.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5905.txt