RF-SITL: A Software-in-the-loop Channel Emulator for UAV Swarm Networks

Nicholas Mastronarde[1], Daniel Russell[2], Zhangyu Guan[1], George Sklivanitis[3], Dimitris Pados[3], Elizabeth Serena Bentley[4], and Michael Medley[4]

[1]Department of Electrical Engineering, University at Buffalo, Buffalo, NY 14260, USA
[2]GE Aviation, Grand Rapids, MI 49512, USA
[3]Dept. of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA
[4]Air Force Research Laboratory (AFRL), Rome, NY 13440, USA
Email: {nmastron, guan}@buffalo.edu, daniel.russell@ge.com, {gsklivanitis, dpados}@fau.edu
{elizabeth.bentley.3, michael.medley}@us.af.mil

***Abstract*—We introduce RF-SITL, a radio frequency (RF) software-in-the-loop (SITL) channel emulator developed with GNU Radio and the University at Buffalo's Airborne Networking and Communications (UB-ANC) emulator to enable integrated simulation of systems comprising multiple unmanned aerial vehicles (UAVs) interacting over a wireless communication channel. RF-SITL could be paired with any multi-robot simulator to enable I/Q sample-level fidelity simulation of communication interactions between the robots by accurately simulating channel effects, including interference, noise, distance-dependent path loss, and packet losses. RF-SITL works as follows: 1) it instantiates a virtual software-defined transceiver in GNU Radio for each UAV simulated in the UB-ANC Emulator; 2) it builds an interference channel model in which each network node receives the superposition of signals transmitted from other nodes; and 3) it synchronizes the location of each simulated UAV in the UB-ANC Emulator with the virtualized RF transceivers in RF-SITL, such that the communication channel between nodes can accurately model distance-dependent channel effects, such as path loss. With these capabilities, we can use both off-the-shelf and custom-built signal processing flowgraphs that simulate Gaussian Minimum Shift Keying (GMSK), 802.11-like Orthogonal Frequency Division Multiplexing (OFDM), and direct sequence spread-spectrum (DSSS) links in GNU Radio to simulate swarm UAV networks prior to their deployment in software-defined radios in a swarm UAV network.**

I. Introduction

Unmanned aerial vehicle (UAV) swarms are poised to enable breakthroughs in a variety of applications including surveillance, emergency first response, package delivery, environmental monitoring, and precision agriculture. Many of these applications include aspects of multi-agent task allocation [1], [2], planning [3], or mapping [4], which require UAVs to take actions collaboratively after interacting with the environment and communicating over a wireless channel. This leads to challenging multidisciplinary problems at the intersection of wireless networking and multi-agent coordination.

Recognizing this, in the last few years, many simulators have been developed to study these challenges, e.g., [5]–[7]. FlyNetSim [5] interfaces two open source tools, namely, ArduPilot [8] and ns-3 [9], to provide synchronized simulation of UAV operations, communications, and networking dynamics. However, it runs as a command line application and does not provide a graphical user interface for visualizing the simulations. We developed the University at Buffalo's Airborne Networking and Communications (UB-ANC) Emulator [7], [10] independently of FlyNetSim. The UB-ANC Emulator interfaces ArduPilot, ns-3 [9], and a ground control station for visualization, such as QGroundControl [11] or APM Planner [12]. More recently, ROS-NetSim [6] was introduced to enable simulation of perception-action-communication loops in multi-agent systems. It can interface with many network and physics simulators, such as ns-3 for network simulation and Gazebo [13] for physics simulation. While ns-3 provides accurate modeling of protocols at the link layer and above, it uses simplistic mathematical models of bit-rates and bit-error rates at the physical layer (see, e.g., the YANS Wi-Fi model in [14]), and does not support I/Q sample-level fidelity transceiver modeling and channel emulation.

Hardware-based channel emulators, such as RFnest [15], enable channel emulation with multiple RF inputs (e.g., radios). However, they can be prohibitively expensive for academic use. Colosseum [16], the worlds most powerful wireless network emulator, is open-access and free to use. It includes 128 standard radio nodes and a massive FPGA-based channel emulator that passes baseband signals through finite impulse response filters representing the channel taps between pairs of standard radio nodes. Although Colosseum supports mobile

ACKNOWLEDGMENT OF SUPPORT AND DISCLAIMER: (a) This material is based upon work supported by United States Air Force under Contract No. FA8750-20-1-0501 and FA8750-20-C-1021; (b) Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

The work of Dr. Dimitris A. Pados and Dr. George Sklivanitis was also supported in part by NSF ECCS-2030234.

Distribution A. Approved for public release: Distribution unlimited AFRL-2022-1528 on 30 Mar 2022

nodes, such as UAVs, node trajectories are fixed and cannot be changed in real-time based on network interactions between nodes. Consequently, it cannot be used for multidisciplinary research at the intersection of wireless networking and multi-agent coordination.

To address the aforementioned challenges and fill the gaps in existing tools, we propose RF-SITL: a radio frequency (RF) software-in-the-loop (SITL) channel emulator developed with GNU Radio and the UB-ANC Emulator. RF-SITL provides I/Q sample-level fidelity simulation and enables a network of virtual software-defined wireless transceivers to experience realistic channel effects. RF-SITL has the following capabilities:

- Instantiate a virtual software-defined transceiver for each UAV that is simulated in the UB-ANC Emulator;
- Develop an interference channel model in which each network node receives the superposition of signals transmitted from other nodes; and
- Synchronize the location of each simulated UAV in the UB-ANC Emulator with the virtual RF transceivers in RF-SITL, such that the channel models between nodes can accurately model distance-dependent channel effects, such as path loss.

With these capabilities, it is possible to simulate UAV swarm networks using any off-the-shelf or custom-built software-defined transceiver architecture implemented in GNU Radio (e.g., Gaussian Minimum Shift Keying (GMSK), 802.11-like Orthogonal Frequency Division Multiplexing (OFDM), and Direct Sequence Spread-Spectrum (DSSS) links) prior to deploying the software-defined transceivers in an actual UAV swarm network. Although we describe how RF-SITL interfaces with the UB-ANC Emulator in this paper, RF-SITL could be paired with any multi-robot simulator to enable I/Q sample-level fidelity simulation of communication interactions.

The remainder of this paper is organized as follows. In Section II, we describe the basic UB-ANC Emulator software architecture and how it interfaces with RF-SITL. In Section III, we design and optimize RF-SITL's architecture. In Section IV, we evaluate RF-SITL's performance and scalability. We conclude and discuss future work in Section V.

II. UB-ANC Emulator with RF-SITL

Fig. 1 provides a high-level diagram of the UB-ANC Emulator's software architecture, which comprises four key components: UB-ANC Agents, software-in-the-loop (SITL) simulators of the flight controller, a ground control station, and RF-SITL.

UB-ANC Agents are implemented using C++ and Qt5 [17]. Each *UB-ANC Agent* represents a simulated UAV and comprises four components: the Agent Control Unit (ACU), the Network Control Unit (NCU), the Micro Air Vehicle Link (MAVLink) Control Unit (MCU), and the Logging Unit (LU). The ACU is the "brains" of a UB-ANC Agent: it contains the application/mission logic and interfaces through well-defined APIs with 1) the NCU to talk with different network elements; 2) the MCU to talk with different flight controllers using the

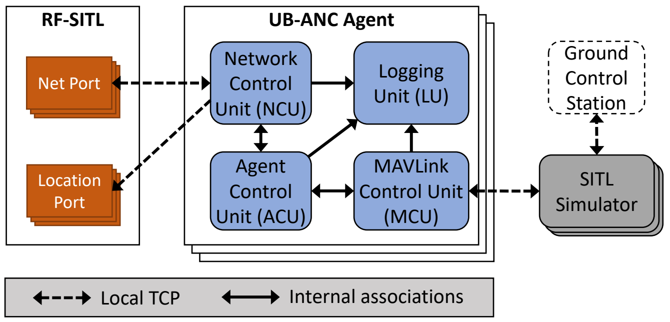

Fig. 1: RF-SITL integrated with UB-ANC Emulator software and SITL simulator of the flight controller.

MAVLink communication protocol; and 3) the LU to log status information. The MCU connects to the open-source Ardupilot SITL Simulator [8], which mimics the behavior of the autopilot code without any autopilot hardware. Finally, the NCU: 1) allows the corresponding UB-ANC Agent to send and receive network packets through RF-SITL using a local TCP port; and 2) sends periodically the corresponding UB-ANC Agent's GPS location to RF-SITL, using another local TCP port, so that RF-SITL can adapt the channel conditions between UB-ANC Agents based on their relative locations (see Section III for more details). Each UB-ANC Agent's SITL Simulator can be connected to an open-source GUI, such as QGroundControl [11] or APM Planner [12], to visualize the emulated UB-ANC Agents.

Note that every UB-ANC Agent, every SITL Simulator, and the ground control station are executed as separate processes. As we discuss further in Section III, RF-SITL can be executed using a single GNU radio flowgraph, which is executed as a single process, or it can be split into multiple GNU radio flowgraphs (one for each UB-ANC Agent), which are executed as separate processes. More information about the UB-ANC Emulator can be found in our prior work [7]. The following section describes RF-SITL in more detail.

III. RF-SITL Architecture

RF-SITL is implemented using Python 2.7.x, GNU Radio 3.7.x, and PyQt4. RF-SITL is executed from a Python script that procedurally generates a flowgraph with N transceivers – each containing the transmit and receive signal paths of a single network node – and constructs an interference channel among them using $N \times N$ channel blocks. Fig. 2 provides a high-level block diagram of RF-SITL. The following subsections provide details about the key components of RF-SITL's architecture.

A. Channel models

RF-SITL connects each of the N transceivers as illustrated in Fig. 2. The link between the output of transceiver n and the input of transceiver m, is associated with one channel model block, denoted by "Channel $n \rightarrow m$" in Fig. 2, where $n, m \in \{1, 2, \ldots, N\}$. We include a self-interference channel from node n's output to its input so that it cannot transmit and receive at the same time in the same frequency band.

Each channel block includes a path loss component that is updated dynamically based on the distance between the corresponding UB-ANC Agents. This is achieved using a custom block that is implemented as an out-of-tree (OOT) GNU Radio module. Specifically, this block collects the GPS coordinates of each node, and translates those to a matrix of channel coefficients that are used to dynamically update the channel taps in each channel block. This is illustrated as the "GPS to Channel" block in Fig. 2.

By default, the channel taps in the channel block between the output of node n and the input of node m are calculated as:

$$c_{nm}[t] = \frac{1}{\sqrt{2}}(1+j) \cdot PL_{nm}[t], \tag{1}$$

$$PL_{nm}[t] = (d_{nm}[t]+1)^{-3}, \tag{2}$$

where $PL_{nm}[t]$ and $d_{nm}[t]$ denote the path loss and distance in meters, respectively, between nodes n and m at time t; and $\frac{1}{\sqrt{2}}(1+j)$ denotes the nominal complex channel coefficient when $PL_{nm}[t] = 1$. Under this model, the time-domain signal received at TXR m can be expressed as:

$$y_m[t] = \sum_{n=1}^{N} c_{nm}[t] x_n[t] + \eta[t], \tag{3}$$

where $x_n[t]$ denotes the transmitted signal from the nth node at time t; $y_m[t]$ denotes the received signal at node m at time t; $c_{nm}[t]$ is the channel coefficient between the nth node and node m at time t as defined in (1); and $\eta[t]$ is zero-mean additive white Gaussian noise at time t with variance σ^2, which is generated by the "Noise Source" block in Fig. 2. To reduce complexity, we use the same additive noise from the "Noise Source" block at the input of all transceivers. Note that this simple channel model can be easily modified. For instance, we can adjust the path loss exponent, introduce a vector of complex channel taps to model multi-path fading, add a non-isotropic antenna propagation pattern to model directional communication, modem I/Q hardware impairments, model signal losses due to airframe occlusion, etc.

B. Hierarchical transceivers

To allow plug-and-play operation, any software-defined transceiver that we want to simulate in RF-SITL is defined in a hierarchical block, denoted by "Hier TXR n" in Fig. 2. The hierarchical block's input is a stream of complex samples that it receives over the simulated interference channel and its output is a stream of complex samples that it transmits over the simulated interference channel. We have implemented and tested three different hierarchical transceiver blocks:

1) A basic Gaussian Minimum Shift Keying (GMSK) based transceiver;
2) An 802.11-like OFDM transceiver based on GNU Radio's OOT module `gr-ieee802-11` [18]; and
3) A cognitive interference avoiding DSSS-based transceiver developed at Florida Atlantic University [19].

Any transceiver-specific parameters can be set through the corresponding hierarchical blocks.

C. Network ports

The nth UB-ANC Agent, $n \in \{1, 2, ..., N\}$, sends and receives data through its corresponding hierarchical transceiver block via the local TCP port

$$\texttt{BASE\_NET\_PORT + 10 * n}, \tag{4}$$

where `BASE_NET_PORT = 15763`. For example, if there are three UB-ANC Agents, then they will send/receive their respective network traffic through ports 15773, 15783, and 15793. This is illustrated in Fig. 2 by the blocks labeled "Net Port (TCP)" in each hierarchical transceiver. Here, data coming "From UB-ANC Agent n" enter the GNU Radio flowgraph, propagate through transceiver n's transmit path, and get transmitted through the simulated interference channel; similarly, data going "To UB-ANC Agent m" are received over the simulated interference channel, processed by transceiver m's receive path, and passed to the corresponding UB-ANC Agent.

D. Location ports

The nth simulated UB-ANC Agent, $n \in \{1, 2, ..., N\}$, hands over its GPS location to RF-SITL via the local TCP port

$$\texttt{BASE\_LOC\_PORT + 10 * n}, \tag{5}$$

where `BASE_LOC_PORT = 15766`. For example, if there are three UB-ANC Agents, then they will send their respective GPS locations to RF-SITL through ports 15776, 15786, and 15796. This is illustrated in Fig. 2 by the block labeled "Location Ports (TCP)."

E. Split RF-SITL architecture

The default scheduler in GNU Radio allows each block to operate on its own thread and the operating system will distribute them across different CPU cores. GNU Radio signal processing blocks read the available samples in their input memory buffer(s), process them as fast as they can, and place the result in the corresponding output memory buffer(s), each of them being executed in its own, independent thread. An underlying runtime scheduler is in charge of managing the flow of data through the flow graph from source(s) to sink(s). As a result, the RF-SITL flowgraph illustrated in Fig. 2 performs a significant amount of signal processing tasks sequentially and requires large amount of time to process each transmitted packet. To overcome this limitation, we adopt hierarchical blocks in RF-SITL, which provide a way to define an arbitrary number of algorithms and implementations (e.g., N separate flowgraphs – one for each UB-ANC Agent – that can execute in parallel) for each processing block, which will be instantiated according to the configuration. Additionally, we create separate threads for controlling the noise source blocks and the distance-based channel model updates to further parallelize the implementation of RF-SITL. The split RF-SITL

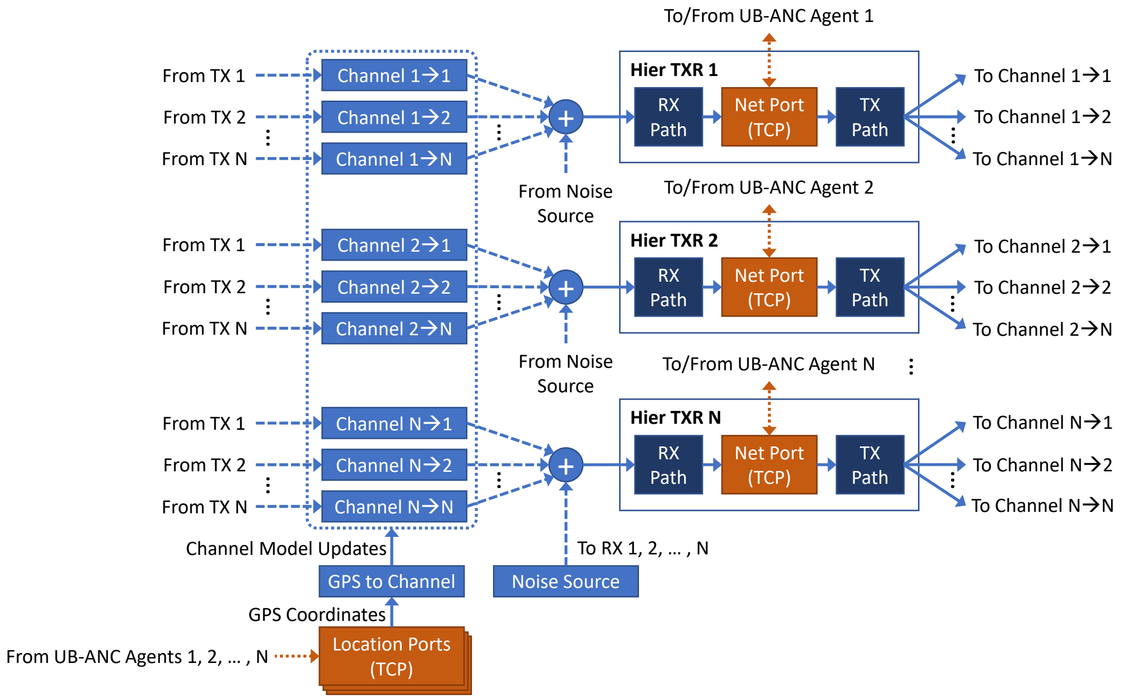

Fig. 2: Centralized RF-SITL architecture with N hierarchical transceiver blocks ("Hier TXR n" for $n \in \{1, 2, \dots, N\}$) interconnected through $N \times N$ channel blocks in a single flowgraph. Channel block (n, m) ("Channel $n \to m$") connects the output of TXR n's transmit path ("TX Path") to the input of TXR m's receive path ("RX Path"). For clarity, these connections are not explicitly shown. Instead, they are indicated by the statements "To Channel $n \to m$" and "From TX n." A single "Noise Source" block generates additive white Gaussian noise at the input of each TXRs' receive path. For clarity, these connections are indicated by the statements "From Noise Source" except at the input of TXR N.

architecture (i.e., the flowgraph built on hierarchical GNU Radio blocks) is illustrated in Fig. 3.

To split the centralized flowgraph, we create custom source and sync blocks in GNU Radio labeled "Signal Subscribe" and "Signal Publish," respectively, in Fig. 3. These blocks take the place of the connectors between the output of the hierarchical transceiver block and the input of the channel blocks in the original centralized flowgraph. The nth transceiver's Signal Publish block, for $n \in \{1, 2, \dots, N\}$, acts as a sink that simply publishes the transceiver's transmitted signal samples to N Unix sockets, with one socket for each outbound channel from node n to nodes $m \in \{1, 2, \dots, N\}$. On the other hand, the nth transceiver's Signal Subscribe block, for $n \in \{1, 2, \dots, N\}$, acts as a source block that reads signal samples from the Unix sockets corresponding to each inbound channel from nodes $m \in \{1, 2, \dots, N\}$ to node n and appends them to buffers associated with each inbound channel. This happens in a separate thread for each channel (and separately from the flowgraph's thread) so it is not blocked by any other processing.

The Signal Subscribe block's "work" function publishes the same number of samples from each buffer to the corresponding channel block. If one of the buffers does not contain enough samples, then the signal is padded with null values $(0 + 0j)$. Additionally, a synchronization mechanism is added to ensure that each transceiver's flowgraph performs the same number of work cycles. Specifically, the nth transceiver's Signal Subscribe block, for $n \in \{1, 2, \dots, N\}$, will send a "ready" signal when it is ready to publish signal samples to each channel block, and then wait for a "trigger" signal to continue. A thread that executes as part of the "GPS to Channel" block listens for these "ready" signals and, upon receiving all of them, sends back a "trigger" signal to every Signal Subscribe block to process a certain number of samples.[1] In this way, each channel processes the same number of samples at the same rate.

IV. Simulation Results

In this section, we demonstrate simulation performance gains obtained using the split RF-SITL architecture, by first comparing the round trip times through the centralized and split architectures with $N = 6$ GMSK transceivers. Subsequently, to illustrate RF-SITL's scalability, we compare the round trip times through the split architecture when there are $N = 2, 3, \dots, 15$ GMSK transceivers. In all simulations,

[1] In the original centralized flowgraph, the "GPS to Channel" block was implemented as a custom GNU Radio block; however, in the split flowgraph, it is just a Python script.

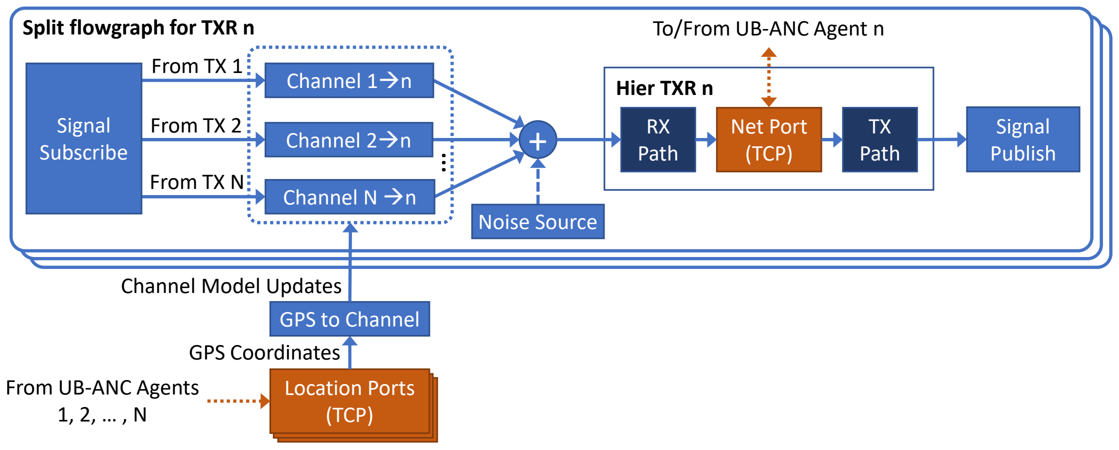

Fig. 3: Split RF-SITL architecture constructed from N flowgraphs that each contain N channel blocks and one hierarchical transceiver block ("Hier TXR n"). The "Signal Publish" block publishes the output of TXR n's transmit path ("TX Path") on a Unix socket. The "Signal Subscribe" block reads the published outputs from TXRs $m \in \{1, 2, \dots, N\}$ and passes them through channel blocks (m, n) ("Channel $m \to n$") to the input of TXR n's receive path ("RX Path"). Each split flowgraph has its own noise source that generates additive white Gaussian noise at the input of the corresponding TXRs' receive path (RX path).

packets have a fixed length of 80 bytes. Simulations take place on a Virtual Box virtual machine running Ubuntu 16.04 LTS with 12 virtual CPU cores (Execution Cap 90%) and 8 GB RAM. The virtual machine is hosted on a computer with an Intel Core i9-9900K 3.60 GHz CPU (8-core, 16-threads) and 64 GB RAM running Windows 10 Pro.

Instead of directly using UB-ANC Agents to send/receive packets through RF-SITL, we created a lightweight COM manager for each transceiver to 1) transmit packets to other transceivers; 2) receive packets from other transceivers and then reply with acknowledgement packets; and 3) measure round trip times. The COM managers for each transceiver are hosted on a separate computer and connected to the machine hosting RF-SITL through a Gigabit ethernet (GigE) switch. Packets transmitted by the COM manager associated with transceiver n are sent through the GigE switch before entering RF-SITL through transceiver n's Net Port. Similarly, packets received by transceiver n in RF-SITL exit through transceiver n's Net Port and are sent through the GigE switch to the COM manager associated with transceiver n.

Fig. 4 illustrates the empirical cumulative distribution functions (CDFs) of round trip times acquired from executing both the centralized and split GNU Radio architectures with $N = 6$ GMSK transceivers (and $N \times N = 36$ channel blocks). Each CDF was generated by measuring the round trip times of 100 packets through the corresponding flowgraph and represents the fraction of these packets with round trip times less than or equal to the round trip time on the x-axis. Since the centralized RF-SITL architecture requires processing to be performed sequentially for every signal processing block and the split RF-SITL architecture enables parallel processing of each transceiver's flowgraph, we observe that the round trip times are smaller when using the latter. The average round trip times measured for the centralized and split architectures are 2.67 seconds and 0.15 seconds, respectively, thus showing that the split architecture is almost 18x faster than the centralized one.

Fig. 5 illustrates the round trip times using the split RF-SITL architecture with $N = 2, 3, \dots, 15$ (and $N \times N = 4, 9, \dots, 225$ channel blocks). For each value of N, we measured the round trip times of 100 packets through the corresponding flowgraph and plotted the average, 10th percentile, and 90th percentile round trip times of these packets. Since the number of flowgraphs increases with the number of transceivers and the split RF-SITL architecture enables parallel processing of each transceiver's flowgraph, the average round trip times increase roughly linearly in N.[2]

From Figs. 4 and 5 it is clear that there are significant variations in the measured round trip times. We were not able to identify the exact cause of these variations, but have several non-mutually exclusive hypotheses. First, the simulations were run in a lab with over 15 networked computers running VNC servers, FTP services, and other applications that may produce network traffic. It is possible that the transit time through the GigE switch between the COM managers and RF-SITL fluctuated because of this traffic. Second, the network interface controller (NIC) on the machine hosting RF-SITL has VNC messages passing through it along with other background network traffic. It is possible that this additional traffic introduced delays. Finally, RF-SITL is executed within a virtual machine and on a host machine sharing resources with other processes. Therefore, it is also possible that the resources available to RF-SITL fluctuated over time.

[2]If there are N transceivers, then each transceiver's flowgraph has N channel blocks as illustrated in Fig. 3. Therefore, the complexity and round trip times are actually superlinear in N.

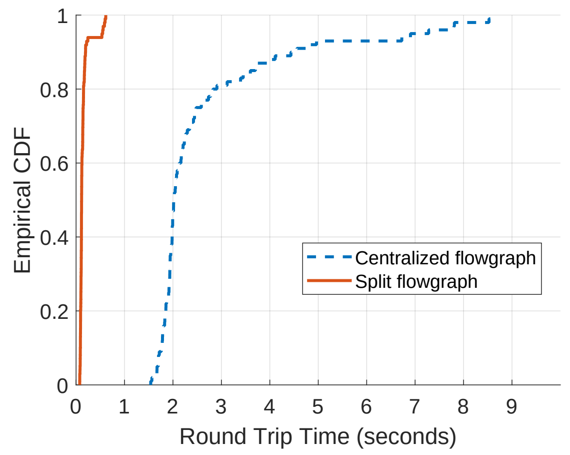

Fig. 4: Empirical CDF of the round trip times in the centralized and split RF-SITL architectures with 6 GMSK transceivers (80 byte packets).

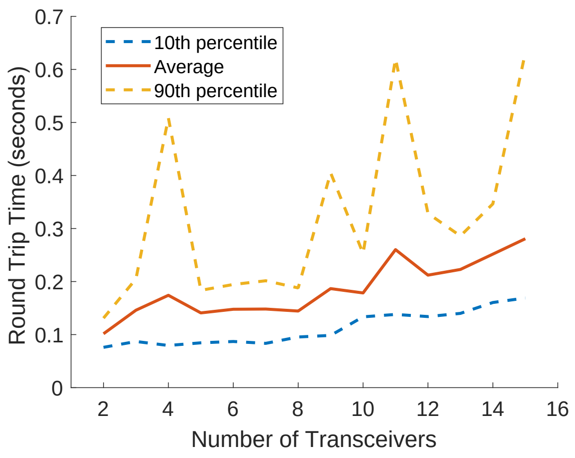

Fig. 5: Round trip times versus the number of GMSK transceivers using the split RF-SITL architecture (80 byte packets).

V. Conclusion

We developed and optimized RF-SITL, a radio frequency (RF) software-in-the-loop channel emulator implemented with GNU Radio and UB-ANC. After implementing RF-SITL with a centralized GNU Radio flowgraph, we observed that it took a long time to process each transmitted packet because of the default design of GNU Radio's scheduler. To overcome this challenge, we adopted a hierarchical design that split the centralized RF-SITL flowgraph into parallel synchronized flowgraphs for each transceiver. Our measurements showed that the split flowgraph was nearly $18\times$ faster than the centralized flowgraph (measured in terms of packet round trip times) in a network with six transceivers. We further demonstrated the scalability of the split flowgraph, which could achieve average round trip times under 300 ms in a network of 15 transceivers. In future work, we will use RF-SITL to compare the effect of different physical layer transceiver architectures on the performance of various multi-agent coordination algorithms (such as task allocation) in the UB-ANC Emulator and compare the results to field tests using the same algorithms and transceiver architectures.

References

[1] M. Rantanen, N. Mastronarde, J. Hudack, and K. Dantu, "Decentralized task allocation in lossy networks: A simulation study," in *16th Annual IEEE Int. Conf. on Sensing, Communication, and Networking (SECON)*, 2019, pp. 1–9.

[2] S. Nayak, S. Yeotikar, E. Carrillo, E. Rudnick-Cohen, M. K. M. Jaffar, R. Patel, S. Azarm, J. W. Herrmann, H. Xu, and M. Otte, "Experimental comparison of decentralized task allocation algorithms under imperfect communication," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 572–579, 2020.

[3] S. S. Ponda, L. B. Johnson, A. N. Kopeikin, H.-L. Choi, and J. P. How, "Distributed planning strategies to ensure network connectivity for dynamic heterogeneous teams," *IEEE J. Sel. Areas Commun.*, vol. 30, no. 5, pp. 861–869, 2012.

[4] M. Pfingsthorn, B. Slamet, and A. Visser, "A scalable hybrid multi-robot slam method for highly detailed maps," in *Robot Soccer World Cup*. Springer, 2007, pp. 457–464.

[5] S. Baidya, Z. Shaikh, and M. Levorato, "FlyNetSim: An open source synchronized uav network simulator based on ns-3 and ardupilot," in *ACM Int. Conf. on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2018, pp. 37–45.

[6] M. Calvo-Fullana, D. Mox, A. Pyattaev, J. Fink, V. Kumar, and A. Ribeiro, "Ros-netsim: A framework for the integration of robotic and network simulators," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1120–1127, 2021.

[7] J. Modares, N. Mastronarde, and K. Dantu, "Simulating unmanned aerial vehicle swarms with the UB-ANC emulator," *Int. Journal of Micro Air Vehicles*, vol. 11, pp. 1–16, 2019.

[8] "SITL Simulator (Software in the Loop)." [Online]. Available: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html

[9] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[10] "UB-ANC Emulator." [Online]. Available: https://github.com/jmodares/UB-ANC-Emulator

[11] "QGroundControl." [Online]. Available: http://qgroundcontrol.com/

[12] "APM Planner." [Online]. Available: https://ardupilot.org/planner2/

[13] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3, 2004, pp. 2149–2154.

[14] M. Lacage and T. R. Henderson, "Yet another network simulator," in *Proceeding from the 2006 workshop on ns-2: the IP network simulator*. ACM, 2006, p. 12.

[15] "RFnest RF Channel Emulator." [Online]. Available: https://www.i-a-i.com/product/rfnest-rf-channel-emulator/

[16] L. Bonati, P. Johari, M. Polese, S. D'Oro, S. Mohanti, M. Tehrani-Moayyed, D. Villa, S. Shrivastava, C. Tassie, K. Yoder *et al.*, "Colosseum: Large-scale wireless experimentation through hardware-in-the-loop network emulation," in *IEEE Int. Symp. on Dynamic Spectrum Access Networks (DySPAN)*, 2021, pp. 105–113.

[17] "Qt Cross-platform software developmenet for embedded and desktop." [Online]. Available: https://www.qt.io/

[18] B. Bloessl, "IEEE 802.11 a/g/p transceiver for GNU radio." [Online]. Available: https://github.com/bastibl/gr-ieee802-11

[19] G. Sklivanitis, A. Gannon, K. Tountas, D. A. Pados, S. N. Batalama, S. Reichhart, M. Medley, N. Thawdar, U. Lee, J. D. Matyjas *et al.*, "Airborne cognitive networking: Design, development, and deployment," *IEEE Access*, vol. 6, pp. 47 217–47 239, 2018.