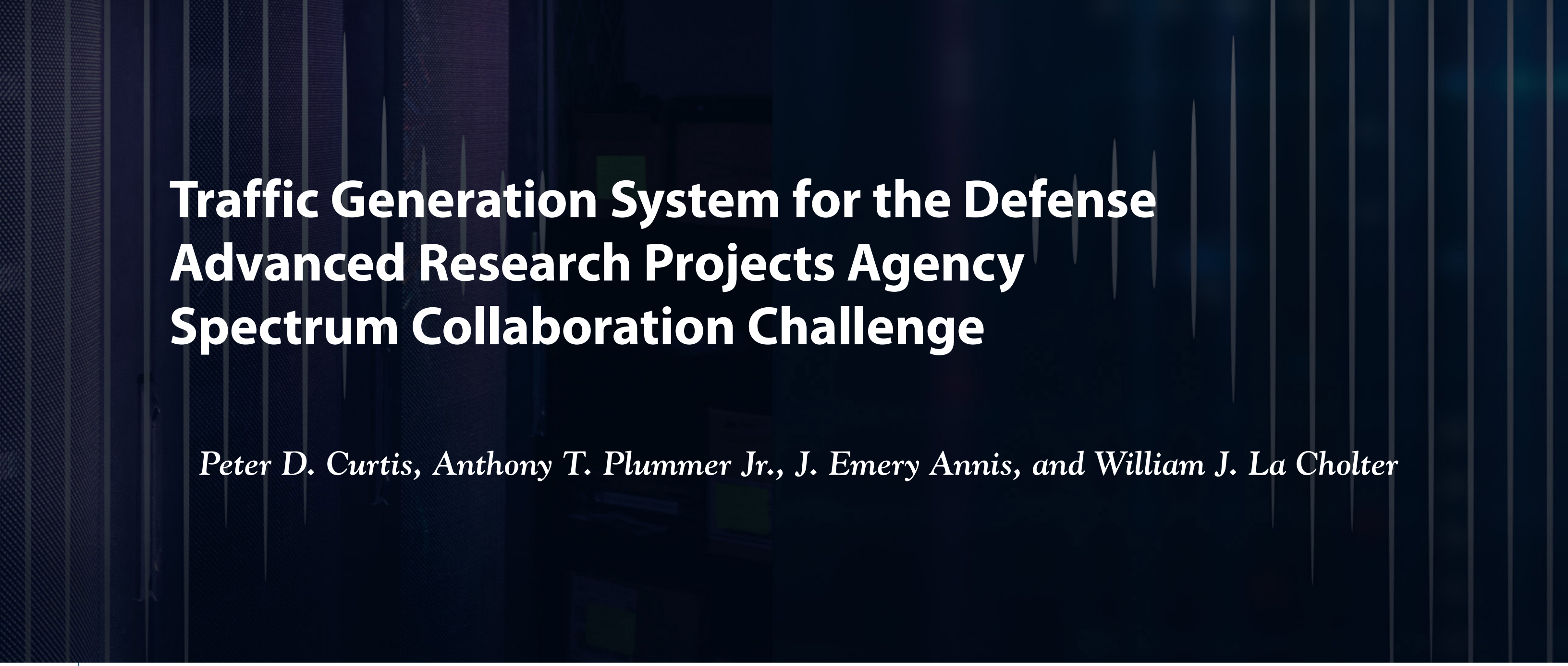


# Traffic Generation System for the Defense Advanced Research Projects Agency Spectrum Collaboration Challenge

*Peter D. Curtis, Anthony T. Plummer Jr., J. Emery Annis, and William J. La Cholter*

## ABSTRACT

*The Defense Advanced Research Projects Agency (DARPA) Spectrum Collaboration Challenge (SC2) required competitors to develop shared spectrum solutions for next-generation communication systems. To enable competitors to test their designs and DARPA to measure and evaluate their utility, the Johns Hopkins University Applied Physics Laboratory (APL) designed and built a wireless research test bed called the Colosseum. One of its components, the Traffic Generation System, enabled on-demand generation and logging of Internet Protocol (IP) version 4 (IPv4) traffic in the Colosseum. The Traffic Generation System simulated a set of network applications running simultaneously on a group of peer nodes, such as a video conferencing application connecting four participants. The Traffic Generation System provided a continuous and unpredictable stream of traffic so that competitors could be measured against a maximum expected traffic flow transmitted through their radios with no possibility of gaining an unfair advantage. IP traffic provides good evaluation metrics because IP packets can be counted, and statistics such as data throughput, latency, jitter, and loss can be calculated. This article discusses the software, hardware, and networking design of the Traffic Generation System.*

## INTRODUCTION

The Defense Advanced Research Projects Agency's (DARPA) Spectrum Collaboration Challenge (SC2) inspired competitors to design solutions enabling radios to collaborate so that they could share, and therefore more efficiently use, the congested radio frequency (RF) spectrum. In support of SC2, APL designed and built a wireless research test bed, known as the Colosseum. (See the article by Coleman et al. in this issue for an overview of the Colosseum.) The test bed resources facilitated research and testing in autonomous spectrum management across a set of collaborative intelligent radio networks (CIRNs). The Colosseum's resources included software-defined radios (SDRs), a wireless channel emulator, emulated backhaul networks, data streams representing realistic user applications, and an emulated GPS service. The Colosseum provided services for research (e.g., a controlled testing environment and secure data storage) and competition (e.g., scorekeeping).

Fundamental to being able to evaluate competitors' designs in the Colosseum was having realistic traffic flows between radios to measure the effectiveness of competitors' algorithms for sharing the RF spectrum. User Datagram Protocol (UDP) over Internet Protocol (IP) version 4 (IPv4) traffic was generated, collected, and measured between each competitor's network of nodes. A competitor's transmitting node received a stream of data that it had to send to a receiving node. That received traffic could be compared against the data originally sent by the source to evaluate the quality of transmission. IP traffic provides good evaluation metrics because each packet can be counted to determine packet throughput, packet latency and jitter, and packet loss. Competitors used these metrics during practice to improve the performance of algorithms and ensure connectivity of the nodes. DARPA used the metrics during competitions to evaluate and compare the performance of competitors' radios.

APL designed and built the Traffic Generation System to enable on-demand generation and logging of IP traffic between CIRNs in the Colosseum. APL custom-built this system, rather than relying on a commercial product, to meet the competition's unique requirements and because of the implementation flexibility offered by a custom system. The Traffic Generation System did, however, use the open-source software Multi-Generator (MGEN) developed by the Naval Research Laboratory (NRL),[1] to generate controlled IP traffic configured with different traffic profiles. When IP traffic was requested for an experiment, the system retrieved the MGEN traffic profile file specified in the experiment configuration and instantiated the MGEN application to generate, receive, and log the transmitted IP traffic between CIRNs. The MGEN application generated unpredictable content, so competitors had to use their radios, rather than caching or predictive algorithms, to ensure that the receiving radios had received all content correctly.

## REQUIREMENTS

Competitors deployed and executed their algorithms on a network of standard radio nodes (SRNs) in the Colosseum. (See the article by White at al. in this issue for more on SRNs.) To evaluate efficiency and collaboration among concurrent users, each competitor SRN was required to exchange data within its network across the RF Emulation System. The Traffic Generation System provided data in the form of UDP over IPv4. The Traffic Generation System provided an application-layer service to SRNs, from which each SRN could prioritize and transmit data streams.

DARPA created several scenarios in which competitors tested, and DARPA evaluated, their designs. These scenarios mimicked real-world situations and obstacles that wireless communications system would face. (See the article by Coleman et al. in this issue for more on the SC2 scenarios.) Scenario files specified application traffic flows, such as video and voice-over-IP (VoIP), as well as profiles for traffic, including steady, bursty, or randomly distributed patterns. Each scenario defined the specific traffic among a set of SRNs. For example, in one scenario a VoIP stream might have had a higher priority than a file transfer stream.

Given the important role of the Traffic Generation System in Colosseum, it had to meet several requirements:

- The system had to emulate IPv4 and UDP.
- The system had to be flexible so that it could generate IP flows capable of emulating specific applications and user network traffic. It had to be possible to specify flows with a given source and destination UDP port, packet size, and packet rate.
- The system had to support the generation of multiple simultaneous flows. Each flow had to be defined with a pair of source and destination SRNs.
- The Traffic Generation System had to be commanded by the Colosseum Resource Manager and support traffic for 128 SRNs simultaneously without any performance degradation. (See the article by Mok et al. in this issue for more on the Resource Manager.)

Each generated packet had the following requirements:

- Packets had to be capable of being marked with Differentiated Services Code Point (DSCP) and type of service (TOS) values for quality of service (QOS) queue processing. Some flows could be transmitted with higher priority than others and were expected to be received without degradation to performance, regardless of the current load on the system.
- Packets had to be tagged with a key-hash message authentication code (HMAC) using keys cryptographically randomly generated and securely supplied to the MGEN application prior to instantiation. This verified that the packet contents could not have been guessed or manipulated, which ensured competition integrity.
- Packets were required to include the transmission timestamp, flow ID, and flow-specific packet sequence numbers. The flow ID was used for mapping flows to SRNs. The transmission timestamp and sequence numbers supported calculating throughput rate, latency, jitter, and loss. All generated traffic had to be easily logged and measured to provide statistics for scoring and visualization systems. Flows had to be logged on a per-packet basis.

- Statistics had to be calculated per flow, per SRN, or per team. All log files and calculated statistics needed to be available to competitors during practice sessions and in scoring and visualization systems during matches.
- Packet payload had be randomized to avoid any caching or compression algorithms and to ensure fairness.

## TRAFFIC GENERATION SYSTEM ARCHITECTURE

The Traffic Generation System delivered on-demand UDP datagrams to SRNs for routing over wireless communication channels between SRNs (through the RF Emulation System; see the article by Barcklow et al. in this issue for more on this system). The datagrams were received back by the Traffic Generation System so it could calculate packet statistics by measuring the offered packet load, packet content, and packet receptions. The Colosseum's primary traffic generator application was MGEN, an open-source IP traffic generator that creates real-time network traffic that can be logged and received for analysis.

To support a large number of simultaneous traffic generation requests with centralized control, the Traffic Generation System had a single traffic controller and multiple traffic generators, each running on a dedicated server. The traffic controller allocated traffic resources, orchestrated traffic allocation and de-allocation, and monitored the health of all traffic generators. Traffic generators provided the traffic resources and managed the collection and distribution of traffic logs. Traffic resources were generated and consumed in Docker containers,[2] each encompassing a single MGEN application for either transmission or reception, and multiple dedicated server devices for hosting the Docker containers. Figure 1 is a diagram of the Traffic Generation System. The Traffic Generation System's innovative design approach enabled a scalable and flexible architecture that could support the diverse requirements of the competition. The system needed to facilitate undefined future traffic profiles, a varying number of simultaneous SRNs traffic flows, and performance targets.

The traffic controller received requests from the Resource Manager during automated (Colosseum-controlled) experiments or from SRNs during manual

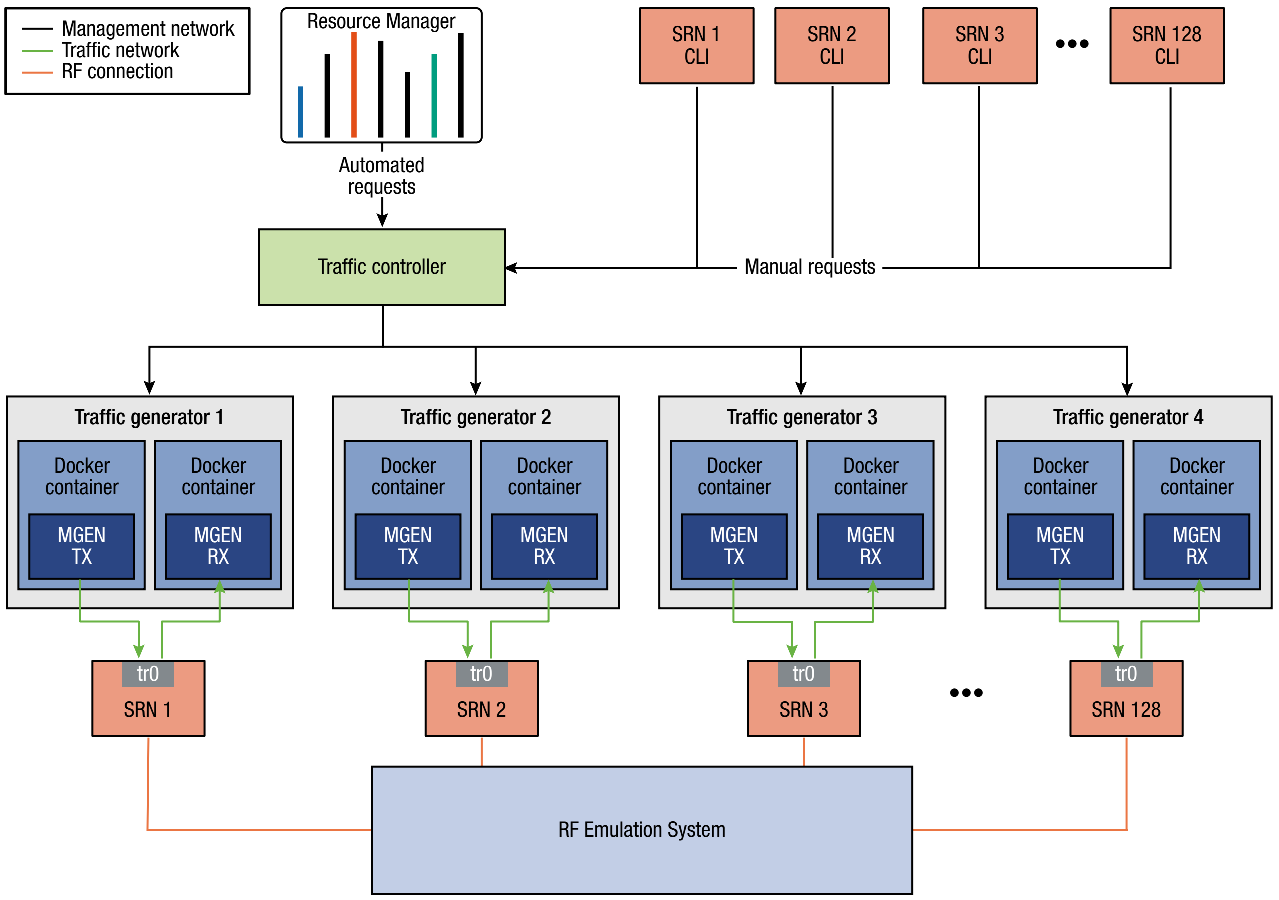


**Figure 1.** Traffic system block diagram. The traffic controller received traffic requests from the Resource Manager or SRNs and determined the appropriate allocation of containers on traffic generation servers. The traffic generators sent IP traffic to be serviced by the SRNs over the RF Emulation System.

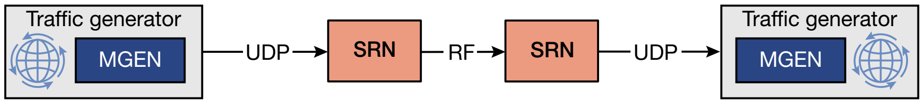


**Figure 2.** Logical traffic generation data path. UDP traffic flowed from an MGEN application in a source traffic generator to a source SRN. The traffic then flowed over RF to a sink SRN and back to a sink traffic generator.

(user-controlled) experiments via a management network. Users on SRNs used the Command Line Interface (CLI) application to send traffic requests (see the article by White et al. in this issue). The traffic controller stored and managed traffic requests and traffic profiles to support experiments. The traffic profiles were defined within a scenario and were aligned to RF channel conditions. These scenarios were preloaded in the traffic controller and were selected by callers to the system. After receiving a request, the traffic controller sent commands to multiple traffic generation servers that managed Docker containers and MGEN applications to generate IP traffic for simulations. Each traffic generator could send or receive IP traffic to any SRN in the Colosseum. The SRNs received traffic via the traffic network on the tr0 interface (see the article by White et al. in this issue) and routed the data over RF through the RF Emulation System.

The start of traffic depended on the mode of operation. In automated mode, traffic resources were allocated and instantiated automatically alongside all other required systems for the automated experiment. The Traffic Generation System alerted the Resource Manager of successful instantiation. When all other systems were prepared in the Colosseum, the Resource Manager sent a second message to initiate the sending of IP traffic to the SRNs. In contrast, during a manual experiment, after receiving a request from a user on an SRN, the traffic controller immediately started the IP traffic once the initialization steps were complete.

Each MGEN instance either generated or received one or more UDP flows between a pair of source and destination SRNs. Figure 2 shows the logical data path for a pair of sending and receiving MGEN instances. Two such send/receive pairs could be used to emulate bidirectional traffic flows. The set of flows between two SRNs was identified as an application. Each application required a transmitter container and a receiving container. When viewing the system from an Open Systems Interconnection (OSI) model[3] perspective, the containers and the MGEN program acted as the application and transport layers, while the SRNs were responsible for routing the offered traffic between the two sending and receiving MGEN containers.

Figure 3 is a diagram of the traffic generation servers and SRN connections. To support its required traffic load, the Colosseum included multiple dedicated traffic generation servers. Each server could support a configurable amount of Docker containers, translating to hundreds of simultaneous traffic applications. Each container was created and destroyed as needed so that system resources could be reused. Each MGEN container was connected to a specific virtual local area network (VLAN) on the traffic servers that were dedicated to a specific SRN. Each sender SRN received its MGEN traffic from its own traffic generation source, and the receiver SRN sent its MGEN traffic to a traffic generation sink. The traffic generation source and sink containers were not required to be on the same server. The figure shows an example traffic flow from a source Docker container, through SRN 1, through the RF Emulation System, through SRN 2, and back to the sink Docker container. The source IP address of an MGEN application was determined by the ID of the corresponding sender SRN, and the destination IP address was determined by the ID of the corresponding receiver SRN. Each SRN was assigned a /24 subnet for traffic generation, allowing for over 200 MGEN sending or receiving instances per SRN. An SRN was connected to traffic generator sources or sinks by a VLAN identified by the SRN's ID. Each VLAN spanned a traffic network composed of 10-Gb networking modules in a blade chassis network backplane and Big Switch software defined networking elements. The Traffic Generation System comprised the traffic network as well as the physical servers and containers on which SRNs and traffic generation sources and sinks resided. The physical servers connected to the traffic network via a 10-Gb connection. The SRNs received traffic on the tr0 interface and routed the data through Universal Software Radio Peripheral (USRP) software defined radios over the RF Emulation System.

## TRAFFIC CONTROLLER

The traffic controller was the service responsible for handling traffic requests. To do so, it had to maintain reservation and traffic scenario state for each session running on the Traffic Generation System. It made the necessary procedure calls to the traffic generation servers to facilitate MGEN traffic flows being sent between the SRNs specified in the request.

The traffic controller was a Python Flask REST (representational state transfer) application programming interface (API) web service[4] running on uWSGI (Unbit

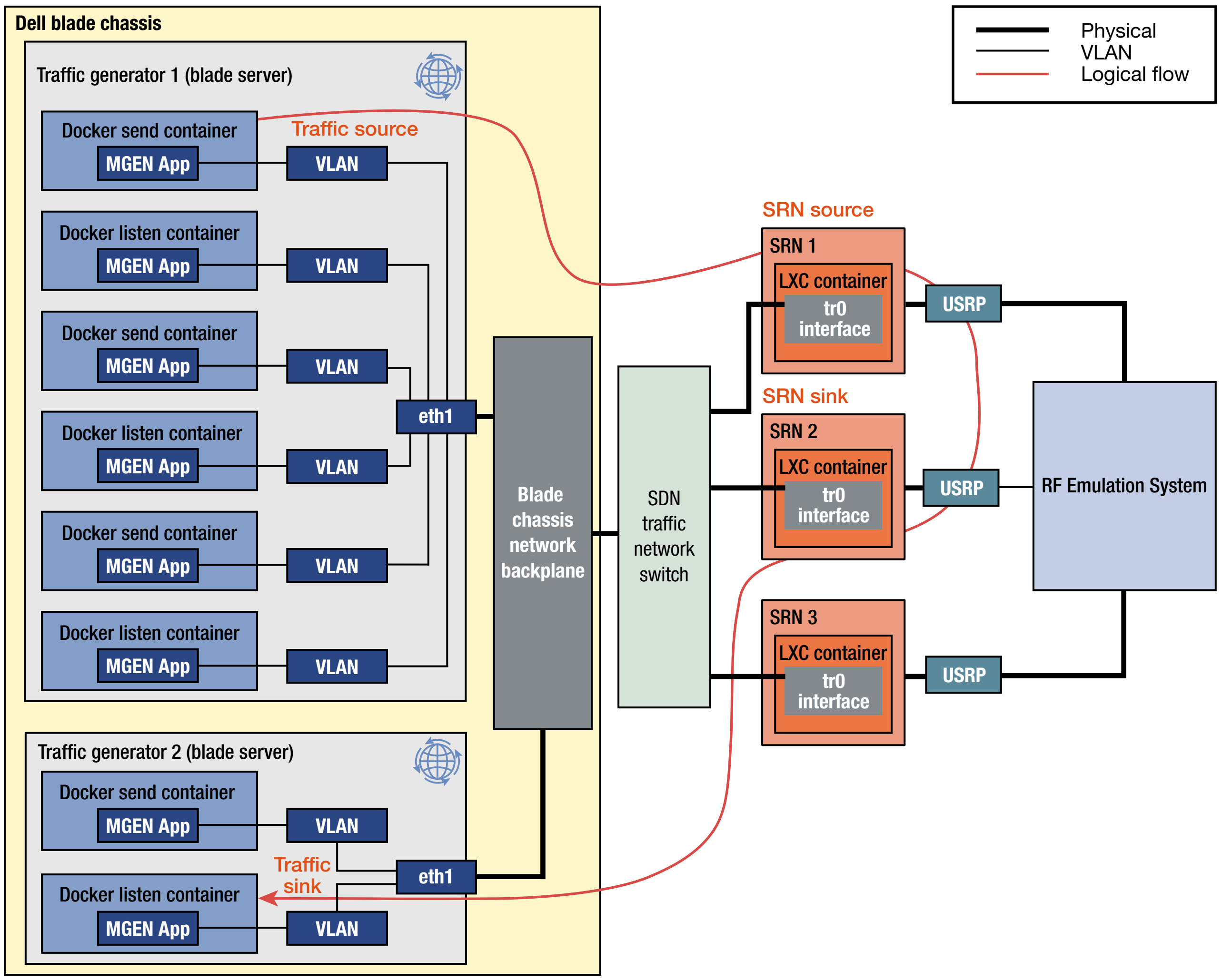


**Figure 3.** Traffic generation servers and SRN connections. The figure shows an example traffic flow from a source Docker container, through SRN 1, through the RF Emulation System, through SRN 2, and back to the sink Docker container.

Web Server Gateway Interface) middleware[5] and an NGINX reverse proxy.[6] The REST API accepted requests to start traffic, terminate traffic, and retrieve traffic request status. Traffic requests for automated experiments were made by the Resource Manager, and traffic requests for manual experiments came from the SRNs.

Each traffic request progressed through a series of states (Figure 4). The general life cycle of a request in the Traffic Generation System was as follows:

1. Requested state
   - A traffic request was received from either the Resource Manager or the SRN CLI.
   - The request's parameters were validated for correctness.
   - The SRNs specified in the request were verified against the Resource Manager's table of active reservations.
   - Allocation algorithms were run (SRN mapping, container allocation, traffic generation allocation).
2. Allocating state
   - The traffic controller commanded the appropriate traffic generators to initialize source and destination Docker containers.
3. Ready state
   - Waited for scenario start message.
4. Active state
   - When a start-traffic message was received (during a manual experiment) or executed by the traffic controller automatically (during an automated experiment), a message was sent to the traffic generators to initiate the start of the MGEN application. This started flowing traffic to the SRNs.

5. De-allocating state
   - A de-allocation request was later received from the Resource Manager or SRNs.
   - The traffic controller commanded the traffic generators to stop MGEN instances, stop and delete Docker containers, and copy MGEN log files to the network-attached storage (NAS) for future analysis.

When scenarios were updated and a scenario reload request was sent, the traffic controller copied updated scenario data directly from the NAS onto the local file system and reran all fixture scripts to populate the database with the updated scenarios.

The process of allocating traffic generation containers involved associating sending and receiving IP addresses, traffic direction, traffic generation host, and session-identifying tokens for each scenario application specified in the node map. The container status, memory, CPU, and throughput statistics were also tracked.

The allocation process began by taking the previously supplied node map and filtering for any scenario applications in the current session that used these nodes in their send or receive positions. Objects were then created holding the aforementioned fields for both send and receive containers (though no traffic generation containers were actually created at this stage).

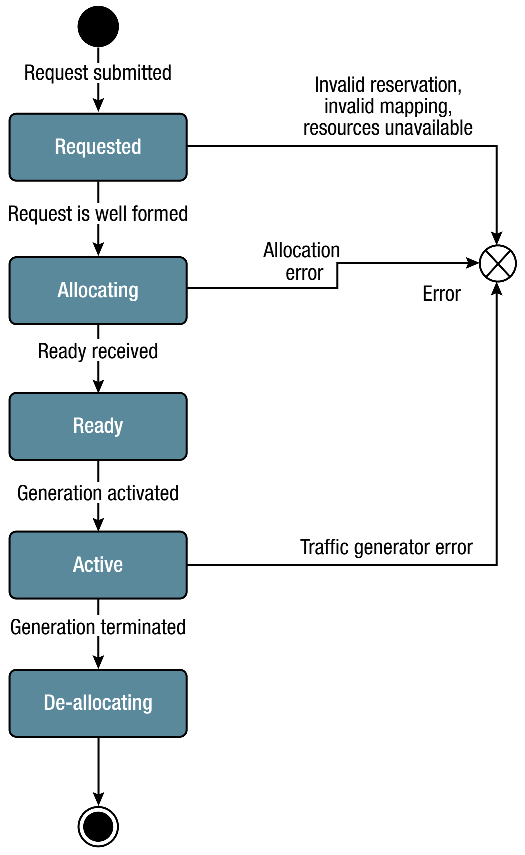


**Figure 4.** Traffic controller session flow diagram. A request through the traffic controller was first validated and verified, then a traffic generator was allocated and activated, and finally after the request completed, the traffic generator was de-allocated.

Before traffic generation containers could be started, each active container had to be associated to a traffic generation server by using a greedy best-fit strategy that chose the traffic generator with the fewest containers allocated to it.

## TRAFFIC GENERATOR

The main purpose of the traffic generators was to host the Docker containers and MGEN applications to generate and receive UDP traffic to and from SRNs, enabling competitors to test their algorithms. The traffic generator, similar to the traffic controller, used a Python Flask REST web service running on a uWSGI middleware and an NGINX reverse proxy. The traffic generator accepted requests to manage Docker containers, including creating and destroying Docker containers and starting and stopping MGEN within the containers. Whereas the traffic controller service managed the lifetime and validation of competitor scenarios and the assignment to resources on all traffic generators, each traffic generator's service managed the resources it hosts, without awareness of competitors.

There were three main phases of an MGEN lifetime, controlled by the traffic generator. A summary of the traffic generator portion of that flow is detailed below:

1. **Receive requests to start Docker containers.** The traffic generator service started the number of requested Docker containers hosting the MGEN service.
2. **Receive request to start traffic.** After the containers were created, a "start traffic" command was received, which triggered the MGEN application to start inside the Docker container. The MGEN application in the sender container would start to send traffic to source SRNs and the listener container would start to receive traffic from destination SRNs.
3. **Receive a stop request for traffic.** When this message was received, the traffic generator stopped the MGEN application running inside the container. The traffic generator then destroyed the container and copied the MGEN log files that stored collected traffic data to an external NAS.

The traffic generator Docker containers were an Ubuntu base image with the MGEN application added. Docker containers were used to enable on-demand and simultaneous creation of traffic generation sources. The traffic generator service interacted with MGEN via an execution interface in the Docker container, sending commands to start or stop MGEN. Docker was managed by the traffic generation software using the Python Docker API library.

Each traffic generator was configured with 128 Linux sub-interfaces to segment the physical Ethernet VLAN (802.1Q) tagged trunk port. Using the Linux Docker container macvlan network driver,[7] 128 corresponding virtual Linux bridges were established. These bridges were used to connect each corresponding sub-interface to any instantiated Docker container. Each Docker network was assigned a subnet ID, gateway IP address, VLAN ID, and parent sub-interface. When a Docker container was created, it was assigned an IP address on this subnet and connected to the Linux bridge. Note that containers were only connected to a single Docker network.

The MGEN application generated, received, and logged all IP traffic. A customized MGEN binary was provided to support payload randomization. This custom binary was developed to offer increased integrity for IP traffic by integrating a key supplied HMAC attached to each packet that could be checked by receiver using a supplied secret file provided by the traffic controller. The MGEN binary also generated a random payload for every packet to prevent packet compression from being used to artificially inflate throughput metrics.

MGEN configuration of traffic flows could be customized by scenario designers in many different ways. MGEN supported the description of IP traffic via command line arguments or through a scripted execution format. The scripted operation supported the high-fidelity traffic requirements of the SC2 competition, so it was integrated into the scenario definitions. MGEN scripts allowed for multiple simultaneous numbered flows to be established within a single application. Flows could then be turned on and off or modified at different times throughout the script.

The Traffic Generation System's ability to log and process traffic statistics was vital to operation. The MGEN application generated log data for each packet sent or received, including flow ID, flow sequence number, source and destination IP, port number, packet size, and timestamp. The NRL-developed TRace Plot Real-time (TRPR) application[8] made it possible to quickly process these logs and plot bit-rate throughput, latency, jitter, and loss.

## TRAFFIC SCENARIOS

A traffic scenario represented a multi-party network application running across two or more network nodes. For example, a traffic scenario might have simulated a UDP-based H.323 application that provided multi-participant video conferencing. The scenario defined the network traffic for each participant and the overall duration of the conference. A competitor's group of SRNs represented the computers used by each participant. The competitor's software on each SRN facilitated data transmission over the RF Emulation System (physical layer) over which the IP-based application had to communicate.

```
{
    "traffic _ scenario _ id": "1",
    "description": "Video, HTTP, and FTP traffic involving 5 Nodes",
    "applications": [
    {
        "app _ id": 1,
        "description": "Netflix between Node 1 and 2",
        "throughput _ max _ bps": 5000000,
        "send _ node _ id": 1,
        "receive _ node _ id": 2,
        "type": "MGEN _ App _ Name",
        "name": "Netflix1"
    },
    {
        "app _ id": 2,
        "description": "Netflix between Node 2 and 3",
        "throughput _ max _ bps": 5000000,
        "send _ node _ id": 2,
        "receive _ node _ id": 3,
        "type": "MGEN _ App _ Name",
        "name": "Netflix1"
    }
    ]
}
```

**Figure 5.** Sample traffic scenario JSON file. Each application described a traffic flow between a sending and receiving node.

```
# _ Start capturing traffic at t=0.0 seconds

0.0 LISTEN UDP 5005

# _ Start the traffic at t=15.0 seconds

15.0 ON 1 UDP SRC 4005 DST dst _ ip/5005 PERIODIC [20 1024]

# _ Stop the traffic at t=86400.0 seconds

86400.0 OFF 1
```

**Figure 6.** MGEN script. A single UDP flow with a source port of 4005, destination port of 5005, sending traffic at 20 messages/second with 1024-byte payload is configured.

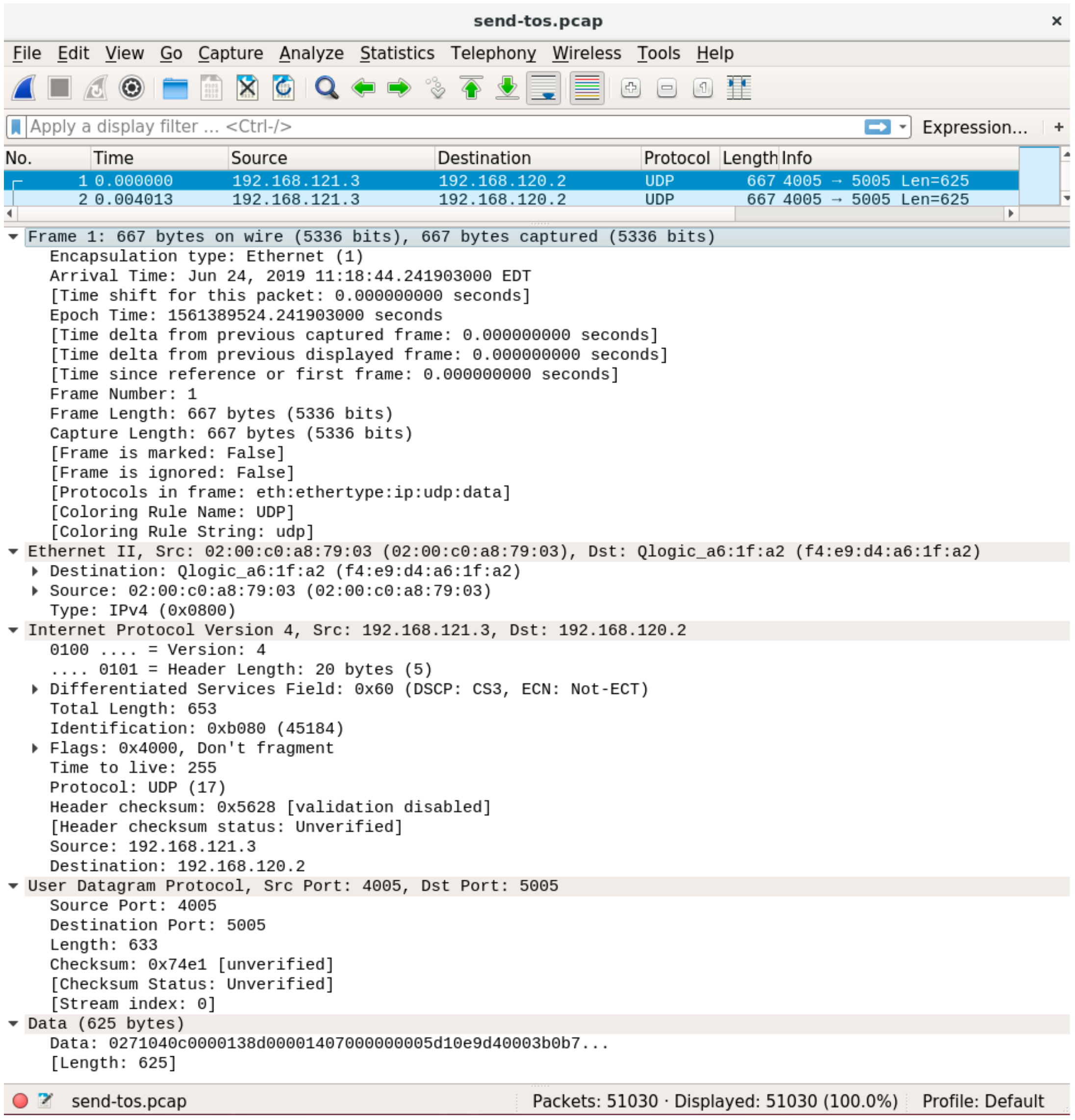
Scenarios were repeatable, but not be predictable—i.e., a competitor should not have been able to record traffic during practice and simply replay it during competition. Scenarios simulated different network traffic bandwidth requirements and profiles, including continuous streams or bursts, lossy or lossless data, and the presence or absence of flow control.

As a complement to RF scenarios, which represented the physical environment in which a group of radios had to operate, traffic scenarios represented the kinds of communications that had to occur in those environments. RF and traffic scenarios together modeled a use case, such as first responders coordinating a response by using mobile radios while they moved through an urban environment that included buildings and lots of background and changing RF traffic.

## Example Scenario

Figure 5 is a partial view of a JSON file for a five-node scenario including three different types of traffic (video, HTTP, and File Transfer Protocol, or FTP). Each application described a traffic flow between a sending and receiving node. The traffic pattern characteristics for each application were described in the named MGEN script files. One MGEN script could describe any number of traffic flows over the duration of the scenario. The MGEN script shown in Figure 6 demonstrates how sending and receiving packets may be configured.

Once the MGEN send command was started, the send MGEN file would start to send UDP traffic from source port 4005 to a destination IP dst_ip to destination port 5005 after 15 seconds. The key dst_ip was replaced with the traffic generator destination IP address


send-tos.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time        Source          Destination     Protocol  Length Info
   1 0.000000    192.168.121.3   192.168.120.2   UDP       667 4005 → 5005 Len=625
   2 0.004013    192.168.121.3   192.168.120.2   UDP       667 4005 → 5005 Len=625
```

```
▾ Frame 1: 667 bytes on wire (5336 bits), 667 bytes captured (5336 bits)
    Encapsulation type: Ethernet (1)
    Arrival Time: Jun 24, 2019 11:18:44.241903000 EDT
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1561389524.241903000 seconds
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 667 bytes (5336 bits)
    Capture Length: 667 bytes (5336 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:udp:data]
    [Coloring Rule Name: UDP]
    [Coloring Rule String: udp]
▾ Ethernet II, Src: 02:00:c0:a8:79:03 (02:00:c0:a8:79:03), Dst: Qlogic_a6:1f:a2 (f4:e9:d4:a6:1f:a2)
  ▸ Destination: Qlogic_a6:1f:a2 (f4:e9:d4:a6:1f:a2)
  ▸ Source: 02:00:c0:a8:79:03 (02:00:c0:a8:79:03)
    Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 192.168.121.3, Dst: 192.168.120.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x60 (DSCP: CS3, ECN: Not-ECT)
    Total Length: 653
    Identification: 0xb080 (45184)
  ▸ Flags: 0x4000, Don't fragment
    Time to live: 255
    Protocol: UDP (17)
    Header checksum: 0x5628 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.121.3
    Destination: 192.168.120.2
▾ User Datagram Protocol, Src Port: 4005, Dst Port: 5005
    Source Port: 4005
    Destination Port: 5005
    Length: 633
    Checksum: 0x74e1 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
▾ Data (625 bytes)
    Data: 0271040c0000138d00001407000000005d10e9d40003b0b7...
    [Length: 625]
```

```
send-tos.pcap        Packets: 51030 · Displayed: 51030 (100.0%)    Profile: Default
```

**Figure 7.** Packet capture. A single packet capture within the flow from source to destination highlights that the IP traffic is IPv4 and possesses the required fields, such as a DSCP TOS value of 0x60.

attached to the destination SRN network. There was a periodic traffic pattern at 20 messages/second with 1,024-byte payload for each message. After 86,400 seconds (or 24 hours), the MGEN traffic stopped. For the MGEN listen command, the container listened on UDP port 5005 immediately and continued until it received a stop request from the traffic generator web service.

A single packet capture within the flow from source to destination highlights that the IP traffic is IPv4 and possesses the required fields, such as a DSCP TOS value of 0x60 (see Figure 7). Following the UDP header is a payload constructed by MGEN that contains more fields to satisfy traffic generation requirements (see Table 1).

## TRAFFIC SERVER PERFORMANCE

The APL team executed a few experiments to evaluate the Traffic Generation System's ability to provide the required IP traffic for the Colosseum. The first experiment aimed to determine the maximum throughput a single Docker container running MGEN can generate. This represented a single application defined in a traffic scenario file. The metric provided scenario developers the limit for a single application flow that could be generated by the Traffic Generation System. A traffic scenario was created with a single UDP flow, 1024-byte packets, and periodic transmit pattern. The packet-per-second rate was varied to increase the offered load in the container. Each run used two Docker containers hosting MGEN applications to send the specified traffic. Figure 8 shows the results of comparing the measured throughput to the expected throughput given the offered load. As shown in the figure, the container was able to maintain the expected throughput of 1,000 and 25,000 packets per second. For larger packets per second, MGEN was not able to sustain the offered load. It was determined that a single container could support approximately 25 MB/s of offered load.

The second experiment sought to determine the maximum traffic a traffic generator server could produce. This metric was important because the number of traffic generator servers could be scaled based on the traffic requirements. Based on the results from the experiment described above, the traffic in a single container was fixed to 25 MB/s and the number of simultaneous maximum-traffic-generating containers was varied. This experiment showed how much traffic a single server could generate while maintaining the offered rate. Figure 9 illustrates the results of measuring the

**Table 1.** MGEN payload fields of a sample packet representing satisfaction of select Traffic Generation System requirements

| Field | Value |
|---|---|
| HMAC | 0x7B377289159B1CAE0554FC530141384A0BECFE97 |
| TX timestamp | 1561389524.24184 |
| Flow ID | 5005 |
| Flow sequence number | 5127 |
| Randomized payload | 548 bytes (entropy of 95.2%) |

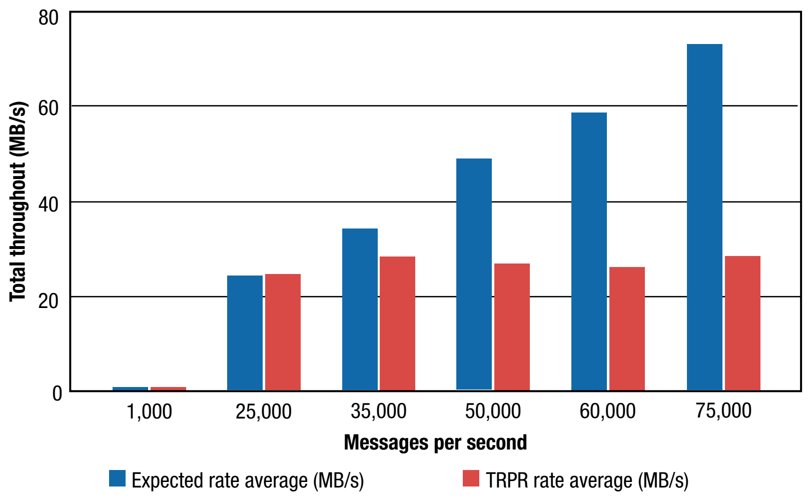


**Figure 8.** Single link throughput per container. A single container was determined to support a maximum of approximately 25 MB/s of offered load.

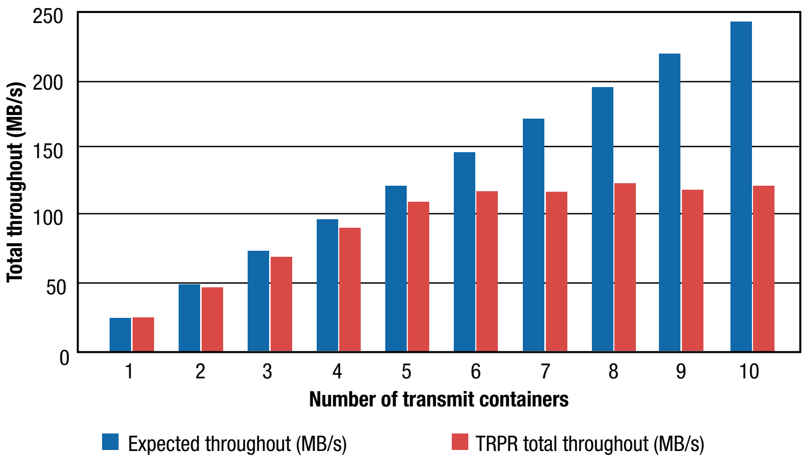


**Figure 9.** Maximum traffic per server. A single traffic generation server was determined to support a maximum of approximately 100 MB/s of offered load.

maximum traffic load per server. The test runs showed that after approximately five containers were transmitting at the maximum rate, MGEN was not able to sustain the offered load. This means that a traffic generator server could support approximately 100 MB/s of offered load.

## CONCLUSION

As part of SC2, competitors developed solutions to share spectrum space effectively and efficiently. To give competitors a test bed for testing their solutions and to give DARPA a way to measure how competitors used and shared spectrum space, APL developed the Colosseum, a large test bed, to support the SC2 competition. The test bed included a custom-built Traffic Generation System that enabled on-demand delivery, reception, and logging of IP traffic for competitors in the Colosseum. IP traffic can provide good evaluation metrics because IP packets can be counted and statistics such as bit-rate throughput, latency, jitter, and loss can be calculated. Throughout the SC2 competition, the Colosseum generated, collected, and measured IP traffic for all competitor radios. This article discussed the software, hardware, and networking design of the Traffic Generation System and reviewed the architecture of the system and how it fit into the Colosseum. It described the traffic network that enabled the transport of IP traffic to radio nodes using an architecture enabled by software defined networking. Finally, it discussed the major software components, the traffic controller and traffic generator.

**ACKNOWLEDGMENTS:** We thank Paul Tilghman (DARPA SC2 program manager) and Craig Pomeroy and Kevin Barone (Systems Engineering and Technical Assistance at DARPA) for their invaluable collaboration and support. We also thank the many APL SC2 contributors, whose names are listed on the inside back cover of this issue of the *Digest*, and in particular we acknowledge Robert W. Grimes and Jeremy G. LaFleur. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US government.

### REFERENCES

[1]"Multi-Generator (MGEN)." US Naval Research Laboratory. https://www.nrl.navy.mil/itd/ncs/products/mgen (accessed Aug. 30, 2019).
[2]"Docker." https://www.docker.com/ (accessed Aug. 30, 2019).
[3]"ISO/IEC 7498-1:1994, Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model." ISO, 1994/1996. https://www.iso.org/standard/20269.html.
[4]"REST API tutorial." https://restfulapi.net/ (accessed Aug. 30, 2019).
[5]"The uWSGI project." uWSGI. https://uwsgi-docs.readthedocs.io (accessed Aug. 30, 2019).
[6]"NGINX." https://www.nginx.com (accessed Aug. 30, 2019).
[7]"Use macvlan networks." Docker, https://docs.docker.com/network/macvlan/ (accessed Aug. 30, 2019).
[8]TRace Plot Real-time (TRPR) application. US Naval Research Laboratory. https://downloads.pf.itd.nrl.navy.mil/docs/proteantools/trpr.html.

**Peter D. Curtis,** Asymmetric Operations Sector, Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Peter D. Curtis is a software engineer in the Communication and Networking Systems Group in APL's Asymmetric Operations Sector. He has a BS in computer science and an MS in computer graphics, both from Mississippi State University. Peter contributed technical expertise to the design and implementation of the Traffic Generation System for the DARPA Spectrum Collaboration Challenge (SC2) Colosseum. His email address is peter.curtis@jhuapl.edu.

**Anthony T. Plummer Jr.,** Asymmetric Operations Sector, Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Dr. Anthony T. Plummer Jr. is the supervisor of the Spectrum Analysis Section in the Tactical Communications Systems Group in APL's Asymmetric Operations Sector. He received a BS in electrical engineering from Morgan State University in 2005 and an MS and a PhD in electrical engineering from Michigan State University in 2007 and 2011, respectively. His interests include the design and implementation of software systems and researching approaches to applying machine learning to communication and networking applications. His email address is anthony.plummer@jhuapl.edu.

**J. Emery Annis,** Asymmetric Operations Sector, Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Emery Annis is a communications systems engineer at APL with a BS in electrical engineering from the University of Houston. He is currently studying for a master's in electrical engineering with a focus in communications from Johns Hopkins University. Emery has contributed to multiple efforts to develop network emulation environments including an Army Wideband Global Satellite (WGS) control system, the DARPA SC2 Colosseum, and an end-to-end networking analysis framework. He regularly contributes to efforts involving RF spectrum coexistence analysis and is interested in applying concepts from both the RF communications domain and IP networking to develop new standards and architectures centered on an intelligent control plane and network maneuver. His email address is emery.annis@jhuapl.edu.

**William J. La Cholter,** Asymmetric Operations Sector, Johns Hopkins University Applied Physics Laboratory, Laurel, MD

William J. La Cholter is a senior computer scientist in APL's Asymmetric Operations Sector. He has a BS in computer science and philosophy from the University of Maryland and an MS in computer science from Johns Hopkins University. Since 2011 he has been a member of the Senior Professional Staff at APL, where he has conducted research and development (R&D) in software diversity, cyber operations, malware attribution, and information security; performed security assessments of US government systems; and developed software for many domains and missions. For the DARPA SC2 program, Mr. La Cholter was a lead for a phase 1 RF Emulation System subteam, developer for the Traffic Generation System, and a software quality subject matter expert. He has led other APL teams as a section supervisor, project manager, and technical lead. Before joining APL, he conducted R&D in law enforcement systems, cross-domain solutions, security incident response, applied cryptography, adaptive networks, firewalls, and high-assurance operating systems. His email address is william.la.cholter@jhuapl.edu.