ANTP Protocol Suite Software Implementation Architecture in Python

Mohammed Alenazi, Santosh Ajith Gogi, Dongsheng Zhang, Egemen K. Çetinkaya, Justin P. Rohrer,
and James P.G. Sterbenz
Department of Electrical Engineering & Computer Science
The University of Kansas
Lawrence, KS 66045
{malenazi,santoshag,dzhang,ekc,rohrej,jpgs}@ittc.ku.edu

ABSTRACT

Due to the highly-dynamic nature of airborne telemetry networks, we have developed the ANTP protocol suite consisting of AeroTP, AeroRP, and AeroNP. Having verified these protocols through simulation and analysis, the next step towards deployment of the ANTP suite is developing a cross-platform implementation of the protocols. Towards this end we present a preliminary architecture for the protocol stack to be implemented in the Python programming language. Initial development and testing is being conducted in the PlanetLab testbed environment, with future trials to be conducted using embedded processors on radio-controlled aircraft and ground vehicles.

I. INTRODUCTION AND MOTIVATION

This paper presents the implementation architecture of the ANTP suite [1, 2]. It is optimized for the highly-dynamic airborne telemetry environment, while maintaining edge-to-edge compatibility with the legacy Internet architecture. The protocols in the suite include: *AeroTP* – a TCP-friendly transport protocol introduced in [3] with multiple reliability and QoS modes, *AeroNP* – an IP-compatible network protocol (addressing and forwarding) introduced in [4], and *AeroRP* – a routing protocol introduced in [4] and further evaluated in [1], which exploits location information to mitigate the short contact times of high-velocity airborne nodes. This protocol suite is designed to perform well in an environment in which the rapidly-changing topology prevents global routing convergence, as well as those in which long-lasting stable end-to-end paths do not exist.

The remainder of this paper is organized as follows: Section II gives the background of the ANTP suite. Section III presents the Python implementation architecture. The details of the modules being implemented are contained in Section IV. Section V concludes and gives future directions for this work.

II. BACKGROUND

This section gives an overview of the protocols found in the ANTP suite, as well as the Python programming language.

A. ANTP Suite

AeroTP is a *TCP-friendly* domain-specific transport protocol designed to meet the needs of the telemetry network environment: dynamic resource sharing, QoS support for fairness and precedence, real-time

data service, and bidirectional communication. AeroTP has several operational modes that support different service classes: reliable, nearly-reliable, quasi-reliable, unreliable connection, and unreliable datagram. The first of these is fully TCP compatible, the last fully UDP compatible, and the others TCP-friendly with reliability semantics matching the needs of the mission and capabilities of the telemetry network. All but the last mode are connection oriented, but are opportunistic and do not use a three-way handshake for connection establishment.

AeroRP is designed to provide reliable communications over a highly dynamic physical topology. The basic operation of AeroRP consists of two parallel phases: neighbor discovery and data forwarding. In the neighbor discovery phase, each node advertises its information to other nodes. Based on the advertised information, each node constructs a routing table containing the position and velocity information of its neighbors. In the data forwarding phase, AeroRP exploits the information present in the routing table to find the most reliable route with the least amount of delivery time. For this purpose, the routing algorithm employs a time metric called the time to intercept (TTI), which determines the least amount of time needed for a neighbor node to reach the packet's destination.

AeroNP is a network protocol designed for highly-dynamic airborne environment. AeroNP provides several functions to make the protocol suite more efficient. To support AeroRP, it carries the location and velocity information of the node and it provides multicast and broadcast functionality. Instead of the IPv4 addressing scheme, AeroNP uses a 2-byte addressing system to reduce the overhead. For AeroTP, it provides a strong header check for a high recovery rate of corrupted payloads.

B. Network programming in Python

C/C++ and Python are very popular languages for network programming. Python requires less than half of the programming time as writing the same program in C, and roughly half the number of lines of code [5]. In C, a careful check of the details including variable lifetimes, memory allocation, header files, and so on are required. The low-level language is preferred if high-performance and portability is required, while a higher-level language expedites the programing process. It is a simply tradeoff between execution speed and developer productivity. Given that we want to have a higher programmer productivity over the machine productivity for the ANTP testbed, Python as a high level scripting language is preferred. The static typing used by some other languages might be better for the purpose of program correctness but reduces productivity. The dynamic typing with minimal scaffolding is a more productive way for Python, though at the risk that a run-time type exception might appear later. Python programmers can focus less on the syntax and grammar of the programming language, but focus more on the realization of the project. Except for the simplicity that is a common characteristic of all high-level languages, Python also keeps its flexibility, an advantage over some other high-level languages [6]. The mandatory indention in Python makes the codes cleaner and easier to read. Although sometimes mixing use of TAB and SPACE will result in confusion for programmer, it helps to prevent bad programming behavior (such as no indention in the next line of an if-clause) from being compiled successfully. Plenty of built-in module choices are available in Python, including many networking blocks, e.g. socket, HTTP, and some web applications, are provided and best supported by the Python Standard Library. In spite of some deficiencies of the high-level language, the advantages of Python over others now make it a popular platform of networking programming, an area in which C has traditionally dominated.

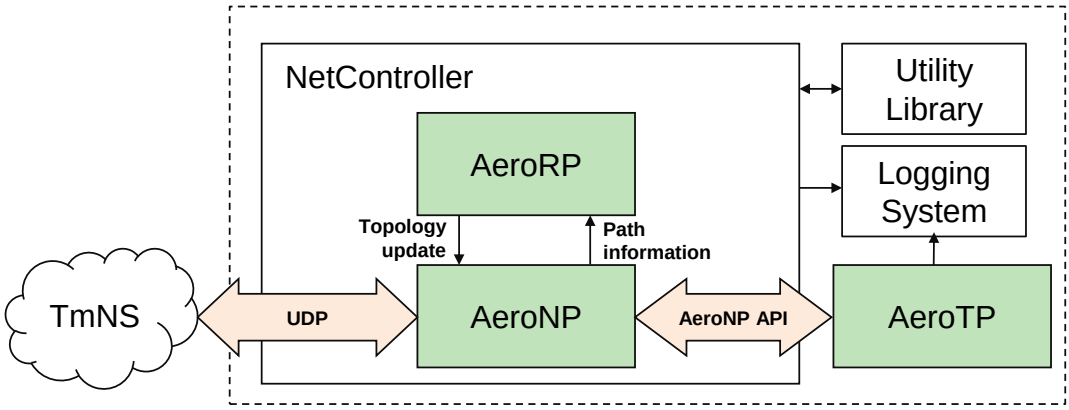

Figure 1: System Architecture

III. SYSTEM ARCHITECTURE

The system architecture is designed to provide several features: maintainability, reliability, and data analysis accessibility. To achieve the maintainability, the system is fully designed based on the object oriented programming (OOP) approach. This approach tries to eliminate the dependency between the data structures, which gives us the option to upgrade the components with less dependency errors. For reliability, the system employs try-catch error handling to avoid any I/O errors during runtime. For performance analysis, the system provides a shared logging system that can aggregate the logs in a single Web server. Based on these considerations, the system is divided into several components as shown in Figure 1.

Utility Library Component. The *Utility Library* consists of several functions essential to the implementation process. The library is shared by all protocols of the system.

- **GPS Location Emulator.** Since PlanetLab nodes are stationary, we present a GPS location emulator to provide GPS-like information for each node. It is implemented as a function that takes an initial position vector P_n^i and a velocity vector V_n of a node i. The position of a node i at time t:

$$P_n^t = (V_n \times t) + P_n^i$$

  This approach provides a way to emulate moving in straight lines. If a node reaches a specified boundary, the node's velocity vector is multiplied by negative one to make it move in the opposite direction. The initial position and velocity vectors are stored in *topology information list*. Each entry in the list is a tuple of three values: the IP address of PlanetLab node, initial position and velocity vector.

- **Binary Functions.** Two functions are implemented to deal with signed binary numbers: *int2bin* and *bin2int*. The *bin2int* function takes two parameters: number and digits. The number is a signed integer value to be converted to binary. The digit is the target number of binary digits that a function should return. The conversation is done based on the $2's$ *complement* of the binary number. The bin2int converts a binary number to a signed integer.

- **Vector Arithmetic.** The routing algorithm is based on the TTI metric that is calculated using basic vector operations such as addition, subtraction and vector normalization. All of these functions are implemented to work with 3-dimensional vectors.
- **Time Representation.** Python has a set of built-in time functions to present and process time in different formats. The reference time used in this implementation is UTC. This decision is made to resolve the problem that PlanetLab nodes are located in different time zones. There are two time formats used in the system; POSIX time and IEEE standard format (RFC 3339). The POSIX time is used in the GPS location emulator and the logging system. The IEEE standard format is used in time stamping the AeroNP packets. The timestamp format is in decimal form; from left to right, the first two digits represent the hour in the range 00–23; the third and fourth digits represent the minute in the range 00–59; the fifth and sixth digits represent the second in the range 00–59. The remaining 4 digits represent the 4 highest-order decimal positions of the second, which have a range of 0000–9999.

Logging System Component is added to check operational correctness and to monitor the performance of the Aero protocols. One way to implement the logging system is to store the logs locally. This is simple to implement, but it would be difficult to aggregate the logs from all the nodes. Furthermore, it would be difficult to monitor and compare the nodes in the real-time. The other approach is to host a database at a Web server and push the logs to the database as created. This approach provides a more convenient way to analyse the performance and check the correctness of the system while running after it is done. Moreover, using a third-party map system, we can present real time visualisation of the running system. For example using the Google Maps API, the main logging web page can show all the running nodes in the system. By selecting a node, more information such as routing table entries will be displayed.

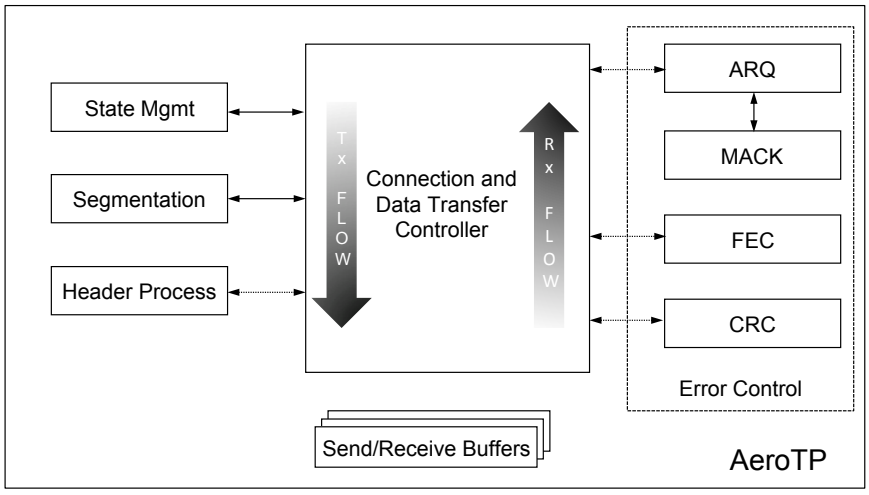

Figure 2: AeroTP implementation architecture

AeroTP Component is designed as a centralized top-level controller module that guides the flow setup, termination and data transfer as shown in Figure 2. In its activity, it employs supplemental modules each designed for a specific service. The controller runs independently on a POSIX operating system environment as a daemon on both source and destination nodes to provide service to the application layer

for end-to-end communication. It utilises services provided by network layer protocol AeroNP in the ANTP suite. The current implementation architecture uses UDP sockets for compatibility with the security restrictions in place on PlanetLab testbed nodes.

NetController Component is a controller program that manages the communication between AeroNP and AeroRP and runs on the operating system as daemon. Furthermore, it creates an API interface to mange the communication between AeroNP and AeroTP, which runs on a separate process. During the initialisation phase, it will start the AeroNP and AeroRP protocols by creating on object of each corresponding class.

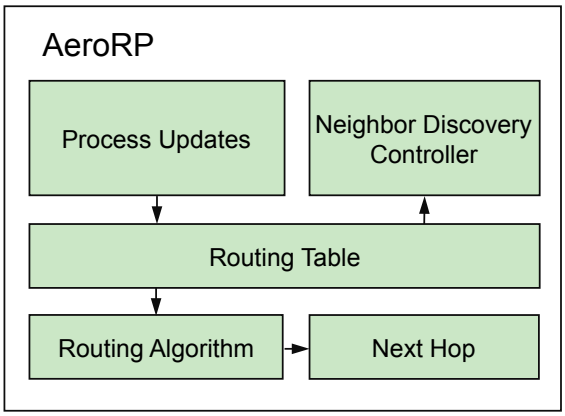

Figure 3: AeroRP component function block

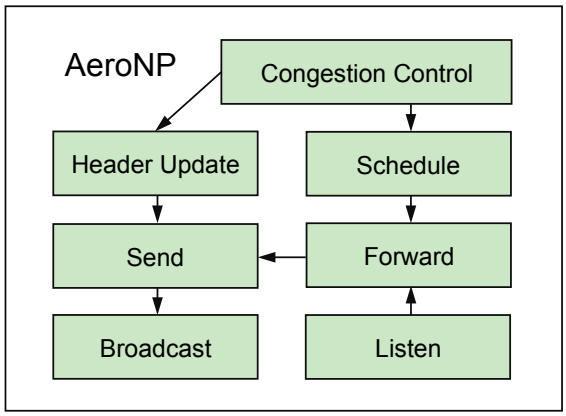

Figure 4: AeroNP component function block

AeroRP Component impliments the AeroRP protocol. It consists of two public functions designed to be used by AeroNP. The first function finds the next hop based on a given distention node. The second function processes updates that fetch incoming AeroNP packets to update the routing table inside the AeroRP component.As shown in Figure 3, AeroRP has other private functions such as the routing algorithm and neighbor discovery controller. The neighbor discovery controller is responsible of advertising the node's location by sending hello messages. The routing algorithm determines the next-hop based on the topology information present in the routing table.

AeroNP Component represents the AeroNP protocol. It has several public communication functions such as send, listen and broadcast. The send function can be used by both AeroRP and AeroTP. For AeroTP, the send function is called through the AeroNP API interface. The listen function can be accessed from the NetController container receive incoming packets. As shown in Figure 4, AeroNP has other private functions designed as helper functions for the main communication functions. For example, congestion control monitors the congestion level of the network. The header update function is used to update the header with the location and velocity information as well as the congestion indicator.

IV. IMPLEMENTATION

This section describes in detail the modular implementation of each of the protocols.

C. AeroTP

The controller flow at the source begins with connection setup that uses the state machine as depicted in Figure 5 in case of the connection-oriented modes [3]. During the connection establishment, the overhead of three-way handshake is eliminated by opportunistic connection approach that overlaps data with control [3]. Segmentation is fallowed with optional FEC error-correcting codes added to the payload. Header processing involves the application layer QoS and error control requirements embedding into `AeroTP_Header` object. Depending on the class of service used by the application, error controlling mechanisms are employed in the subsequent steps. These functions are complimented appropriately at the receiver by to decompose the segment based on service mode, maintaining state, and aggregating ACKs fallowed by connection teardown. Beside these modules, a consistent and efficient buffer-handling mechanism augments the main controller using send and receive buffer.

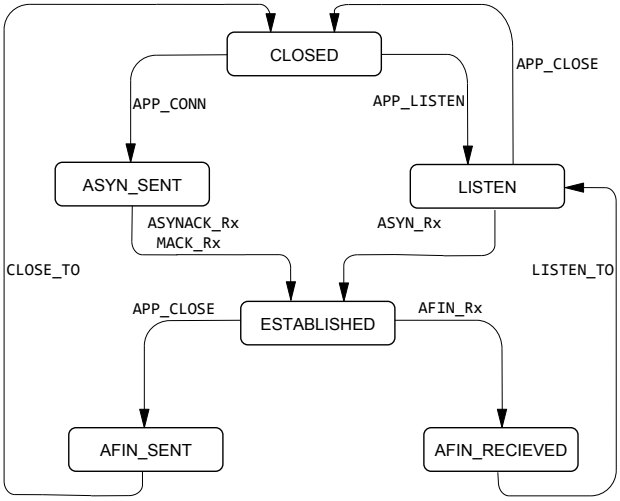

Figure 5: AeroTP state machine

While the centralised controller provides the main functionality, each supplemental module provides a key piece of solution to the application *error control* requirements. These modules make use of the object-oriented features of Python. These modules act as independent libraries, instantiated by the centralised controller.

Segmentation and Header processing modules are a prerequisite for processing related to size and alignment of control and payload data during network communication. The segmentation module accepts data and segments it to chunks to be inserted into `SEND_BUFF` buffer list. In contrast, the receiver side decapsulates the segment from `RECV_BUFF` buffer list. Segmentation also provides the sequence numbering mechanism, which numbers each transport protocol data unit to permit resequencing.

Header processing at the source helps in constructing the header of 16 byte (in network byte order) using the binary packing library method `struct.pack(format, v1, v2, ...)`, which can easily be decomposed at the destination. The arguments `v1,v2,..` that are header fields are packed according to `format`. This module provides extensive support in terms of handling header field attributes.

State Management module effectively manages the state transitions in the case of connection-oriented

end-to-end communication. This is an event-driven module that transitions based on an event occurrence from `CURRENT_STATE` to `NEW_STATE` and takes appropriate action. The events are listed in Table 1. It maintains AeroTP states and possible event transitions with action handlers.

Table 1: State Transition Definitions

State	**Description**
CLOSED	State in which no connection exists and no data is transferred
LISTEN	State in which a receiver is ready to listen to any incoming data
ASYN_SENT	ASYN message sent by the host initiating connection
ESTABLISHED	Steady state in which data transfer takes place
AFIN_SENT	AFIN message sent to indicate no new data being sent
AFIN_RECEIVED	AFIN message received and AFINACK is sent as an acknowledgement
APP_CONN	Request issued by the application to initiate connection by sending ASYN
APP_LISTEN	Request issued to the receiving host to move to the LISTEN state
APP_CLOSE	Request to initiate closing a connection by sending AFIN
ASYN_RX	ASYN received, indicating a connection has been requested
ASYNACK_RX	ASYNACK received, indicating connection request has been granted
MACK_RX	Single or multiple packet ACK received
AFIN_RX	AFIN received, indicating end of any new data, initiating connection close
CLOSE_TO	A timeout allowing outstanding retransmissions before going to CLOSED state
LISTEN_TO	A timeout to go to LISTEN state so that the receiver can receive all data packets

Error-control modules ARQ, FEC, and CRC provide different types of reliability to the service classes. ARQ (automatic repeat request) is employed in the reliable mode of AeroTP [7]. Positive acknowledgement in combination with the payload CRC is used to guarantee the effective data delivery. For the AeroTP protocol, the selective repeat ARQ algorithm is used to provide reliable edge-to-edge connection between nodes. In our Python implementation, ARQ is built as a module which can be optionally used by the central controller. There are four AeroTP modes and ARQ is only used in the reliable mode. A pair of AeroTP client and server sockets are employed. A separate Python thread using `thread.start_new_thread(function, args)` is created with function timer for each sending segment. By using the `socket.settimeout(value)` of the socket library, a value of the timer is set in the sending end, and failing to receive the ACK from the receiver within the timeout will trigger the retransmission of the lost packets. The thread will be closed automatically once the sender has confirmed the successful delivery of that segment. The acknowledgment mechanism of ARQ, namely MACK (multiple ACK) dynamically aggregates many ACKs depending on the transmission rate and loss rate at the destination end.

FEC (forward error correction) is based on open-loop error recovery mechanisms, eliminating ARQ and ACKs entirely. This provides an arbitrary level of statistical reliability but without absolute delivery guarantees. FEC uses the Reed-Solomon forward error correcting based library [8] that represents the payload as polynomials and encoding at the source. The forward error correcting code object is created by the constructor with two arguments `n` and `k` where `n` is the length of a codeword and `k` is the length of the message, must be less than `n`. This has the ability to correct at most $(n-k)/2$ errors. The source encodes the payload, which on receipt by the receiver decodes to correct the errors if existing. This module can be

tuned to provide error correcting strengths between 0 and 100%. Though FEC adds additional redundancy of inserting codes into the payload, it overcomes the need to retransmits.

The CRC (cyclic redundancy check) protects the integrity of the data end-to-end across telemetry network. It can be used in conjunction with FEC to detect errors. The HEC (header error check) is a 16-byte strong CRC where as the payload has a 32-byte CRC performed by the Python library module `binascii`. AeroTP also supports hybrid error control in which FEC provides statistical reliability, with ARQ invoked only as occasionally needed for full reliability.

D. AeroRP

The AeroRP class uses three other classes. The main class of AeroRP is the routing table class. Before presenting the routing table class, we will present the other two because they are used in implementing the routing table.

- **GeolocationTuple** is represented as a class. The main function of this class is to represent a node inside the routing table. It maintains several pieces of information about a node: IP address, location, velocity, range and recoding timestamp. Also, it maintains a list of neighbors represented as *NeighborTuple* class.
- **NeighborTuple** is also implemented as class but it has less information than *GeolocationTuple*. The main function of this class is to represent the neighbors of each node. It maintains IP address, time start, time expire and link cost.
- **RoutingTable** is a main component of the AeroRP. It maintains a list of the nodes in the network represented as GeolocationTuple.

The AeroRP class has two public functions designed to be used by AeroNP clases:

- **next–hop** is a function that determines the next hop for a given destination address. It performs the routing algorithm based on the TTI metric system.
- **processUpdates** is a function that is designed to be called by AeroNP. Incoming packets will be received by the listen function inside AeroNP class. One a packet is received, a copy of it is sent to this function, which checks if the packet has any routing table updates.

E. AeroNP

The AeroNP class uses one of two packet formats to represent a single AeroNP packet: basic header and extended header. The basic header format is designed to be relatively compact whereas the extended header carries optional location and trajectory information that is utilized by the AeroRP routing protocol.

- **Basic Header and Extended Header** are implemented as classes. The values are stored as binary-like format using packed structures. The main goal of using this approach is to match the packet length, using some functions in the Utility Library. For each field of the header, there are two accessors to set and get the value of that field. To get the binary representation of the header, each class has a `get-datagram` function.

The AeroNP class has five functions. It has three public functions send, broadcast and listen. These functions are designed to be used by AeroRP and AeroTP. The other two functions are private that are designed are called by the public functions:

- **send** function encapsulates AeroTP and AeroRP packets into an AeroNP packet. Then, based on the destination address, the next-hop address is obtained from the AeroRP class using the `next-hop` function. The AeroNP payload is obtained by the `get-datagram` function of AeroNP packet. The next-hop address, obtained from the `next-hop` function, is assigned a destination address for the UDP socket and the payload is sent over the UDP socket.
- **broadcast** is a main network layer function needed by the AeroRP. Since we are using just selected nodes of PlanetLab, we have to implement a broadcast function that uses unicast sending functionality. Basically, the function sends a given AeroNP packet to all the nodes in *the topology information list*.
- **listen** is a function that is implemented as an infinite while loop. A UDP socket is created and set to listen for incoming packets. If the packet is of extended type header, a copy of the packet is passed to the function `ProcessUpdates` that is located in the AeroRP class to update the table based on the information located.
- **forward** is a private function that can be called by the `listen` function. Once a packet is received from `listen` function, it is passed to the AeroTP protocol if destination source is the local host address and the protocol ID is the corresponding AeroTP ID.
- **headerUpdate** is a private function that can be called by the `listen` function. It updates the AeroNP packet header. It accepts the two types extended and basic. Based on the type, it inserts the geolocation information generated by the GPS location emulator into an AeroNP extended header packet when the packet is sent from the current node. Also, it sets the congestion indicator that is obtained from the congestion controller.

F. AeroNP API

The AeroNP API is an AeroNP interface designed to be used by the AeroTP protocol. It uses UNIX sockets to listen to AeroTP requests. The socket has two ends; one is accessed by the AeroNP and the other is accessed by the AeroTP. Each end has two functions `send` and `readTPbuffer`. For the AeroNP end, the functions are `send2tp` and `readNPbuffer`. The buffer is a FIFO list in Python. For the AeroTP, the functions are `send2np` and `readNPbuffer`. AeroTP can send a packet by passing the payload to the `send2np` function. Then, the function is passed to the `send` function, which is in the AeroNP class, to be sent to the destination node. Once an incoming AeroTP packet is checked in AeroNP, it is inserted into the buffer. Using the size attribute of the buffer, each end can detect if there are packets to be sent or received.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an overview of our design architecture for implementing the ANTP protocol suite. We have given our rationale for using Python as the reference implementation language, as well as discussing the design considerations to allow maximal portability, including PlanetLab compatibility.

As we continue the implementation project, we will add additional variables to the testing environment, such as mobile nodes, which will change some of the prototyping requirements as well as adding realistic challenges as we cross-validate our implementation results against past simulation results.

ACKNOWLEDGEMENTS

The authors would like to thank the Test Resource Management Center (TRMC) Test and Evaluation/Science and Technology (T&E/S&T) Program for their support. This work was funded in part by the T&E/S&T Program through the Army PEO STRI Contracting Office, contract number W900KK-09-C-0019 for AeroNP and AeroTP: Aeronautical Network and Transport Protocols for iNET (ANTP). The Executing Agent and Program Manager work out of the AFFTC. This work was also funded in part by the International Foundation for Telemetering (IFT). We would like to thank Kip Temple and the membership of the iNET working group for discussions that led to this work.

REFERENCES

[1] J. P. Rohrer, A. Jabbar, E. Perrins, and J. P. G. Sterbenz, "Cross-layer architectural framework for highly-mobile multihop airborne telemetry networks," in *Proceedings of the IEEE Military Communications Conference (MILCOM)*, (San Diego, CA, USA), pp. 1–9, November 2008.

[2] J. P. Rohrer, A. Jabbar, E. K. Çetinkaya, E. Perrins, and J. P. Sterbenz, "Highly-dynamic cross-layered aeronautical network architecture," *IEEE Transactions on Aerospace and Electronic Systems (TAES)*, vol. 47, October 2011.

[3] J. P. Rohrer, E. Perrins, and J. P. G. Sterbenz, "End-to-end disruption-tolerant transport protocol issues and design for airborne telemetry networks," in *Proceedings of the International Telemetering Conference*, (San Diego, CA), October 27–30 2008.

[4] A. Jabbar, E. Perrins, and J. P. G. Sterbenz, "A cross-layered protocol architecture for highly-dynamic multihop airborne telemetry networks," in *Proceedings of the International Telemetering Conference (ITC)*, (San Diego, CA), October 27–30 2008.

[5] L. Prechelt and C. Java, "An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program,"

[6] M. Gordon, "An introduction to network programming the python way [book review]," *Distributed Systems Online, IEEE*, vol. 6, no. 10, 2005.

[7] K. S. Pathapati, J. P. Rohrer, and J. P. G. Sterbenz, "Edge-to-edge arq: Transport-layer reliability for airborne telemetry networks," in *Proceedings of the International Telemetering Conference (ITC)*, (San Diego, CA), October 2010.

[8] A. Brown, "Reed solomon class of error correcting codes in python." `https://github.com/brownan/Reed-Solomon`.