Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations Student Graduate Works

3-2022

An Entity-component System Based, IEEE DIS Interoperability Interface

Noah W. Scott

Follow this and additional works at: https://scholar.afit.edu/etd

Part of the Software Engineering Commons

Recommended Citation

Scott, Noah W., "An Entity-component System Based, IEEE DIS Interoperability Interface" (2022). *Theses and Dissertations*. 5358.
https://scholar.afit.edu/etd/5358

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.

AN ENTITY-COMPONENT SYSTEM BASED, IEEE DIS INTEROPERABILITY INTERFACE

THESIS

Noah W Scott, 1LT, USAF

AFIT-ENG-MS-22-M-060

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-22-M-060

AN ENTITY-COMPONENT SYSTEM BASED, IEEE DIS INTEROPERABILITY INTERFACE

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Noah W Scott, B.S.C.E
1LT, USAF

March 24, 2022

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-22-M-060

AN ENTITY-COMPONENT SYSTEM BASED, IEEE DIS INTEROPERABILITY INTERFACE

THESIS

Noah W Scott, B.S.C.E
1LT, USAF

Committee Membership:

Douglas D Hodson, Ph.D.
Chair

Richard Dill, Ph.D.
Member

Michael R Grimaila, Ph.D., CISM, CISSP
Member

AFIT-ENG-MS-22-M-060

Abstract

In practice, there are several different methods of organizing data within a given software to fulfil its function. The method known as the Entity-Component System (ECS) is a software architecture where data components define entities. These components are stored as organized lists which are operated upon by systems to inject the system's desired behavior. The standard of IEEE Distributed Interactive Simulation defines a set of protocol data units (PDUs) that are exchanged between networked simulations within a distributed system. The standard also defines specifics regarding how each simulation sends, receive,s and processes the PDUs. The standard supports military simulation application communication across a multitude of local and distanced locations.

When leveraging a network to exchange PDUs, each simulation can represent a common understanding of the world, at the desired level of detail, to allow hosted entity interactions. DIS-compliant simulations are commonly written using an Object-Oriented Programming paradigm, where data is contained within objects and state is altered through exchanged method calls (or messaged) through a publicly defined interface. This research investigates the viability of an ECS-compliant IEEE DIS interoperability interface by creating and testing a prototype application that can send, receive, and process Entity State PDUs as defined by the standard, allowing it to interact with pre-existing simulation technologies.

A prototype, written in the Rust programming language, incorporates the nalgebra-glm library (i.e., Crate) to provide general-purpose linear algebra features to perform the mathematical operations associated with prediction (i.e., dead reckoning) and convergence; the Serde library to serialize and deserialize PDUs; and the Handy ECS

(hecs) library to provide a minimalist entity-component-system (ECS) capability. Our prototype implements a simple cyclic scheduler loop to execute the ECS systems that updates the components.

The results of this research are that creating an ECS-based DIS simulation node is viable. It is also possible to implement the functionality of Dead Reckoning and DIS requirements using the Rust programming language.

AFIT-ENG-MS-22-M-060

To my wife, when times felt tough and nothing seemed to go right, you were always there to support me.

Acknowledgments

I would like to thank my advisor, who helped me find motivation even in a pandemic. I'd also like to thank everyone whom I've been able to meet through AFIT, you've all played a role in helping myself and others at the institution to succeed.

Table of Contents

Page

Abstract iv

I. Introduction 1

1.1 Problem Background 1
1.2 Research Objectives 1
1.3 Background 2
1.3.1 Rust 2
1.3.2 Serde 3
1.3.3 Distributed Interactive Simulation (DIS) 4
1.3.4 IEEE DIS Standard 5
1.4 Document Overview 5

II. Paper I: Using Serde to Serialize and Deserialize PDUs 7

III. Paper II: A Data-Oriented ECS-based Design to Send DIS Packets 12

IV. Paper III: Prediction and Convergence Calculations using Rust-based NAlgebra-GLM 17

V. Paper IV: An ECS-Based DIS Interoperability Interface 23

VI. Conclusions 34

6.1 Future Work 34

AN ENTITY-COMPONENT SYSTEM BASED, IEEE DIS INTEROPERABILITY INTERFACE

I. Introduction

1.1 Problem Background

The Distributed Interactive Simulation (DIS) is an IEEE standardized protocol implemented and used to connect military simulations across multiple geographically distanced locations. Applications for this space have traditionally used an Object-Oriented C++ approach for design. However, this is not the only possible approach. There has been no published research which explores the viability of using a Data-Oriented Design for DIS simulation applications. Entity-Component Systems, a type of Data-Oriented Design, is a common implementation for private sector gaming applications, so we hypothesize that a similar approach could assist in improving the data management and organization of our own simulation capabilities.

1.2 Research Objectives

- Create a data-oriented ECS environment capable of sending DIS-compliant packets
- Implement Dead Reckoning Algorithms in Rust that function within the ECS
- Construct an ECS-based interface for DIS Interoperability that can exchange EntityState packets

1.3 Background

The following sections are details regarding technologies and standards that went into creating this research software prototype for an ECS-based DIS interface.

1.3.1 Rust

Part of the nuance of this research effort is achieved by using the Rust programming language. Rust is originally a language created by Mozilla for use in their browser Firefox, specifically to operate multi-threaded shared data applications. It caught the attention of C++ developers due to its advantages regarding efficiency, memory management, and safety restrictions built into the compiler.

With regards to C++, Rust is a significantly younger language, having only been utilized in published code since 2010 compared to the published release of C++ in 1985. However, Rust is built to be more intuitive than C++, noted for features such as preventing unwanted behavior and producing more specific error messages when issues arise when building and executing code. Rust is also quickly becoming more prominent in the private sector gaming world due to these qualities, so pursuing progress in our own simulation domains using the language is an opportunity to acquire cutting-edge technology.

The key to Rust's memory management is that the language does not allow for bad pointers (such as dangling pointers) or null pointers. This means that issues such as memory leaks are inherently prevented when coding with Rust, both during development and at run time. Rust is also strict regarding data ownership, requiring that any given piece of data can only be "owned" by one part of the software at a time, and and attempt to alter the data after transferring ownership will result in an error. Rust is still capable of taking these "unsafe" actions, but only if written using the "Unsafe" portion of Rust. The language is essentially divided into one part

that is “Safe” and the other part that is “Unsafe”, and both can be used in the same program. However, as we have no need to write Unsafe code for this research (and really should be trying to avoid it in general), all of the software development for this effort is written with Safe Rust, eliminating the worry regarding data issues. It is this safety combined with the DIS goal of real-time (or near real-time) applications that makes Rust a uniquely appropriate language to research for application into the simulation environment.

1.3.2 Serde

A requirement for networked applications is the ability to exchange data over a network. However, generally entities or objects within any given application cannot be sent in the same form that they are stored in. Instead, they must be serialized, a process which converts data within a program into a more network-compliant form. Then, the receiving application must deserialize the new form back into the original pieces (which reverses the Serialization process). This process may be accomplished in multiple ways, but for this research we are leveraging the processes provided by the Serde Crate (or package).

Serde can serialize and deserialize a wide range of data types. A defined type in Rust, for example, would be able to be serialized as long as the type is predefined with serialization tags. The prototype software can construct the PDU locally, serialize it using Serde, then send it over the network using its network interfacing system. Upon receiving a packet, Serde can also perform the reverse, deserializing the packet back into the original PDU that was sent from the source node. By providing this functionality, the software is able to communicate in the same manner as DIS applications, which is one of the intended goals of the research development.

1.3.3 Distributed Interactive Simulation (DIS)

DIS has two meanings. IEEE DIS refers to the standards that are found in the IEEE documentation for DIS. The other type would refer to the basics regarding a distributed simulation.

Any type of distributed simulation has to have certain basic functionalities. The first is that the simulation is able to be properly executed in near real-time (as little lag time as possible between communications) over multiple distributed locations, likely spanning hundreds of miles in-between at a minimum. Coders who engineer the simulation software build it with this fact in mind, as certain potential functionalities will either degrade or completely break down if not done over a Local Area Network (LAN).

DIS also has a specific method of operation in the network space because the system is designed to function without a central server. Instead, all DIS simulations are made up of nodes which exchange data between one another in order to run the simulation. This method of operation is part of what allows all players in a DIS environment to have roughly equal latency as there's no one player closer to the server than the others. However, this approach incurs a performance cost.

Commonly, DIS works over a UDP connection which is considered an unreliable protocol since messages are not intrinsically acknowledged by the intended receiver. As a result, there is a chance that a packet may be dropped by the network and the receiving node will not be aware of the information in the packet, potentially causing a simulation action to not be properly recorded. This method is used due to versatility, as one can have a UDP system with X total users, and they can all still receive the packet. Otherwise, each node would have to establish a TCP connection with each other node (TCP is a connection between 2 systems which requires acknowledgement packets) to ensure the data was properly received on each individual line. This slower

method would unfortunately be detrimental to a key function of DIS, which is to support real-time interactive simulations.

Each DIS system has certain standards that it follows to ensure each node can properly communicate with others on the same simulation network. For our application, we follow the IEEE standard.

1.3.4 IEEE DIS Standard

As stated previously, a DIS simulation must be standardized for each node that joins, ensuring that all connected players have a properly functioning simulation experience. The IEEE DIS Standard is the standard which most directly applies to our intended application for this research: Air Force simulation environments. The main function of this documentation is to standardize PDU formatting. It does not defines each type of PDU utilized in any given DIS standard, but it lays out the usage and data contents that should be found in each PDU type. By documenting and standardizing these intricacies, the IEEE DIS Standard ensures that all software developed for use in a standardized simulation will be sending and receiving data in the proper form. As a result, all software applications can be developed asynchronously from one another and still interact properly over the distributed environment.

1.4 Document Overview

This thesis is a compendium of four papers written during the research. Chapter II contains the first paper which is the initial work to assemble a DIS-compliant PDU and send it over the network. Chapter III contains the second paper and provides the development of the framework for the full interface. Chapter IV contains the third paper which introduces the design to fully implement DIS functionality. Chapter V contains the fourth paper which provides implementation details of the proof-of-

concept research interface. Finally, Chapter VI presents the research conclusions and future recommendations.

II. Paper I: Using Serde to Serialize and Deserialize PDUs

The following paper, "Using Serde to Serialize and Deserialize PDUs," was published in Fall 2020 and presented at CSCI 2020.

Using Serde to Serialize and Deserialize DIS PDUs

Noah W. Scott, Douglas D. Hodson, Richard Dill, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
email: noah.scott.meta@gmail.com, doug@sidechannel.net, richard.dill@afit.edu, michael.grimaila@afit.edu

Abstract— *Serialization is the process of translating a data structure into a format that can be stored and/or transmitted, and then subsequently reconstructed at a later time to create an identical clone of the original. The use of data serialization assures data objects can be transmitted, stored, and reliably reconstructed across differing computer architectures, even with different data type sizes or endianness, with no additional effort.*

Serializing the data in an architecture-independent format prevents the problems of byte ordering, memory layout, or representing data structures in different programming languages. This is especially important in the context of live, virtual, and constructive (LVC) simulation environments where multiple geographically separated computers, each with many independent threads, are connected and must communicate with as little latency as possible to remain near "real-time" like in terms of responsiveness.

In this paper, we demonstrate the use of Serde, a Rust-based systems programming language crate, to serialize and deserialize IEEE standard Distribute Interactive Simulation (DIS) Protocol Data Units (PDUs) to support DIS-based network interoperability. The results show that Serde is an efficient mechanism for serialization/deserialization when using the inherently safe Rust programming language.

Keywords: Simulation Distributed DIS PDU Rust Serde bincode

1. Introduction

Data serialization is a fundamental capability that software engineers must implement so that simulations of many independent, and geographically distributed application can share data objects over network interfaces. Applications use data structures to define state and efficiently communicate changes that need to be processed by other applications that compose the distributed system.

In the domain of distributed interactive simulation, these applications are often written in a systems-level programming language, such as C or C++, which provide low-level, high-control for how to organize and present data to support network operations in addition to excellent execution performance. These attributes are a result of the languages being "closer" to the hardware level and require the programmer to handle memory management, threads, garbage collection, and other critical tasks that can place the safety and integrity of the application at risk if not done properly. In contrast, the Rust programming language provides a robust type system and ownership model that guarantees memory-safety and thread-safety, which alleviates that dynamic at compile-time. Rust is fast, memory-efficient, and easily integrates with other languages.

This research examines using the Rust system-level programming language in the domain of distributed interactive simulation. Since Rust compiles code that is guaranteed to be "thread safe," this supports our goal to eliminate data races associated with shared data being updated and used between multiple executing threads. This is especially important in military-oriented flight simulators designed to operate in a distributed networked environment that typically create and execute multiple independent threads to service different functional aspects of the application (e.g., model updates, graphics, networking, data logging). A good introduction to military-style distributed simulation can be found here [1].

One especially appealing aspect of the Rust programming language is that it enables less experienced developers to write complex multi-threaded code with some level of guaranteed correctness. Code written that might cause a data race simply does not compile. This reason and motivation to evaluate it is nearly identical to the reason Mozilla states as a motivation for the creation of the language itself. In effect, identifying thread-based errors at compile time improves overall product quality.

2. Background

The next couple of sections provide an overview of the technologies used to perform this work.

2.1 DIS & PDUs

Distributed Interactive Simulation (DIS) is an IEEE standard [2] for conducting real-time platform-level wargaming across multiple host computers and is used worldwide, especially by military organizations but also by other agencies such as those involved in space exploration and medicine [3].

To facilitate "interoperability" among individual network-compatible applications (i.e., "simulations"), this standard defines the data packets to be exchanged, so-called Protocol Data Units (PDUs), and how to interpret them. This paper does not delve into all of the details associated with the standard, but focuses on one specific PDU, the Entity State PDU. Entity State PDUs are nearly ubiquitous in the sense it is understood by virtually all DIS-compliant simulations. For

a good introduction to DIS, see the Simulation Interoperability Standards Organization [4] (SISO) published document titled "DIS Plain and Simple. [5]"

2.2 Rust

The Rust [6] systems-level programming language is a low-level, high-control language that, much like C and C++, does not include a garbage collector. By leveraging a rich type system in conjunction with a memory ownership system, many classes of bugs can be caught and eliminated at compile time. In other words, classes of problems that are avoided by experienced C++ programmers through knowledge and experience (and maybe documented in detailed coding practices and standards), do not pass the compilation stage of the Rust compiler.

The Rust compiler (by default) enforces code to be compiled to obey rules associated with data mutation. When programming in this way (i.e., "safe"), all data races that might occur due to coding errors that would otherwise plague a multi-threaded application are completely avoided, as the code will be rejected by the compiler. The Rust compiler cannot perform magic, it cannot detect logic errors or deadlocks (threads waiting on each other), but it certainly helps eliminate entire classes of problems.

Rust can be thought of as a combination of two programming languages: "safe" Rust and "unsafe" Rust. Conveniently, these names mean exactly what they say: safe Rust is safe. Unsafe Rust is, well, not. In fact, unsafe Rust lets programmers do some really unsafe things. Safe Rust is the true Rust programming language. If programmers code only using safe Rust, you will never have to worry about type-safety or memory-safety. You will never endure a dangling pointer, a use-after-free, or any other kind of undefined behavior [7]. For this effort, we only use "safe" Rust.

Since distributed interactive simulations inherently form a set of real-time networked multi-threaded applications, the Rust software engineering features are uniquely suited to this domain. Although different than the initial motivations from the author, Mozilla, parsing cascading style sheets, the use case is essentially the same: the reliable creation of multi-threaded applications that share data.

2.3 Serde

One functional aspect of creating these networked applications is the serialization and deserialization of DIS-based PDUs to/from a network. "Serialization" is defined as the process of turning some Rust-based structure into a data format that can be restored later. "Deserialization" is defined as the reverse of that process: taking data structured from some format, and rebuilding it into a Rust-based structure.

A framework for how to translate data in one format to another is exactly what the popular crate (Rust's terminology for a library or package), Serde, is designed to do. Specifically, Serde is a framework for **serializing** and **deserializing** Rust data structures efficiently and generically [8].

The Serde ecosystem consists of data structures that know how to serialize and deserialize themselves along with data formats that know how to serialize and deserialize other things. Serde provides the layer by which these two groups interact with each other, allowing any supported data structure to be serialized and deserialized using any supported data format.

2.4 bincode

`bincode` [9] complements the Serde framework by providing serialization to/from a binary data format - specifically, it defines a compact encoder / decoder pair. For a Rust structure that can be serialized or deserialized with Serde, bincode encodes it to a vector of bytes (unsigned 8-bit values) and decodes the vector of bytes back to the Rust structure. It's a perfect complement to Serde's ecosystem to support encoding Rust-based DIS PDU structures for sending or receiving to a stream of bytes, that are then sent or received from a network device.

2.5 KDIS

In order to test the Rust-based, Serde compliant serializer and deserializer, we collected a network encoded PDU using Wireshark [10] generated from a reliable source that can produce well formed, DIS-compliant PDUs with the definable fields that the IEEE DIS standard specifies. We selected the popular open-source KDIS [11] package to perform this function.

KDIS is a "a complete open source implementation of DIS (Distributed Interactive Simulation) in C++. 1278.1 and 1278.1a are fully implemented including enumerations [11]." The package also includes several example applications to demonstrate its use.

3. Entity State PDU Header

DIS PDUs are assembled in two pieces, a header which defines metadata-like information concerning the version of the protocol, information associated with the "exercise" (i.e., the collection of simulations communicating), the type of PDU being sent, among other things, followed by the PDU itself. The main aspect of concern, the header, consists of 96 bits (12 bytes) of information. Figure 1 shows a Wireshark captured PDU.

Fig. 1: Entity State PDU

As shown in Figure 2, the decoded header indicates this PDU conforms to the IEEE 1278.1-1995 version of the standard, and is for a Entity State PDU (PDU type 1). In later versions, some of the 16 bits associated with the "padding" field is used to define a "status" field. At this time, we are only concerned about this version of the standard.

```
Wireshark · Packet 3 · Wi-Fi
> Frame 3: 186 bytes on wire (1488 bits), 186 bytes captured (1488 bits) on i
> Ethernet II, Src: Tp-LinkT_19:e6:8b (08:1f:71:19:e6:8b), Dst: Shenzhen_60:d
> Internet Protocol Version 4, Src: 192.168.10.194, Dst: 192.168.0.255
> User Datagram Protocol, Src Port: 60096, Dst Port: 3000
v Distributed Interactive Simulation
  v Header
      Proto version: IEEE 1278.1-1995 (5)
      Exercise ID: 1
      PDU type: Entity State (1)
      Proto Family: Entity information / interaction (1)
      Timestamp: 30:45.459698 (relative)
      PDU Length: 144
      Padding: 0000
  > Entity State PDU

0000  78 a3 51 60 d9 63 08 1f  71 19 e6 8b 08 00 45 00
0010  00 ac b9 f7 00 00 80 11  f3 37 c0 a8 0a c2 c0 a8
0020  00 ff ea c0 0b b8 00 98  0d fc 05 01 01 01 83 3b
0030  91 8c 00 90 00 00 00 01  0b b9 00 03 01 00 03 01
0040  00 e1 03 00 01 00 03 01  00 e1 03 00 01 00 00 00
0050  00 00 00 00 00 00 00 00  00 00 c1 43 ee fb 20 5c
0060  5b 1d c1 4f 22 1c 96 73  17 37 41 4f 8b 5f eb 25
0070  c8 30 3f 80 2a 73 bf 5c  6f 74 c0 49 0f db 00 00
0080  00 00 01 00 00 00 00 00  00 00 00 00 00 00 00 00
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00a0  00 00 00 00 00 00 00 00  00 00 01 4b 41 52 4c 00
00b0  00 00 00 00 00 00 00 00  00 00

Close   Help
```

Fig. 2: Entity State PDU Header Details

For this research, we are also only concerned about defining this header as a Rust data structure and writing the code needed to serialize and deserialize it using Serde. Defining every aspect of the entity state PDU can follow, but, for this proof-of-concept, this is all that is needed.

4. Application Design

A Rust-based application was written that depends upon the Serde and bincode crates. The proof-of-concept application itself was about 50 lines long and defined the DIS PDU header structure as shown in Figure 3.

```
// IEEE 1278.1-1995
#[derive(Serialize, Deserialize)]
#[derive(Debug)]
#[repr(C)]
struct PDUHeader {
  protocol_version: u8,
  exercise_identifier: u8,
  pdu_type: u8,
  protocol_family: u8,
  time_stamp:u32,
  length:u16,
  padding: u16,
}
```

Fig. 3: PDU Header Structure

The Rust attributes associated with the structure (identified by "#" sign) indicate that Serde is to derive (i.e., generate) implementations for the Serialize and Deserialize traits, Rust is to derive an implementation to print the structure for debugging, and lastly, Rust should represent (i.e., layout) the data structure in memory as the "C" language would.

Because we are compiling and executing this application on a little endian PC architecture, Rust's built-in functions to swap bytes were used for both the "time_stamp" and "length" fields. All other bytes are single byte values and do not need any adjustment.

5. Testing & Verification

A vector of bytes was defined to include what is shown highlighted in Figure 2. This sequence of bytes is considered the encoded (or serialized) representation for the PDU header structure.

```
let encoded: Vec<u8> = vec![0x05, 0x01, 0x01,
                            0x01, 0x83, 0x3b,
                            0x91, 0x8c, 0x00,
                            0x90, 0x00, 0x00];
```

bincode can serialize this sequence starting from a struct instance, but for our purpose, we want the test to be more direct (the sequence of bytes is the encoded data). bincode is then used to deserialize this sequence to a PDU header struct.

```
let mut decoded: PDUHeader
   = bincode::deserialize(&encoded).unwrap();
```

Each field of the structure was then printed to the console which revealed an identical match with what Wireshark shows in the top window of Figure 2.

Opening a UDP socket and sending the data is trivial, given that Rust's standard library provides platform independent support to do that. Chanda's book [12] titled "Network Programming with Rust" provides numerous examples of how to do this.

6. Final Thoughts

Using Serde in conjunction with bincode's binary implementations for the serialize and deserialize methods provides a powerful, efficient means to stream DIS-based simulation state to/from a network device. Although not discussed, Rust's built-in networking features also makes it easy to open network sockets for sending or receiving data of interest.

Using Serde opens up other opportunities as well, such as the easy serialization of the DIS PDU's to/from JSON, YAML, MessagePack, TOML, BSON, and many more.

7. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] D. D. Hodson and R. R. Hill, "The art and science of live, virtual, and constructive simulation for test and analysis," *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, vol. 11, no. 2, pp. 77–89, 2014.

[2] "IEEE Standard for Distributed Interactive Simulation – Application Protocols," *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*, pp. 1–747, 2012.

[3] "Distributed Interactive Simulation," 2020, accessed 15-Oct-2020. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_Interactive_Simulation

[4] "Simulation Interoperability Standards Organization," 2020, accessed 15-Oct-2020. [Online]. Available: https://www.sisostds.org/

[5] "Reference for guide: DIS plain and simple," *SISO-REF-020-DRAFT*, 2007.

[6] "Rust website," 2020, accessed 15-Oct-2020. [Online]. Available: https://www.rust-lang.org

[7] "The Rustonomicon," 2020, [Accessed 15-Oct-2020]. [Online]. Available: https://doc.rust-lang.org/stable/nomicon

[8] "Serde," 2020, accessed 15-Oct-2020. [Online]. Available: https://docs.serde.rs/serde

[9] "Bincode," 2020, accessed 15-Oct-2020. [Online]. Available: https://github.com/servo/bincode

[10] "wireshark," 2020, accessed 15-Oct-2020. [Online]. Available: https://www.wireshark.org

[11] "KDIS," 2020, accessed 15-Oct-2020. [Online]. Available: https://sourceforge.net/projects/kdis

[12] A. Chanda, *Network Programming with Rust*. Birmingham, UK: Packt Publishing Ltd., 2018.

Author Biographies

NOAH W. SCOTT is a 1st Lieutenant in the United States Air Force, currently stationed at Wright Patterson Air Force base. He is attending the Air Force Institute of Technology, where he is studying the use of software engineering in modeling and simulations solutions. He is a 2018 graduate of the Missouri University of Science and Technology, where he studied computer engineering and computer science. His previous duty station was with AFRL's 711th Human Performance Wing to develop software training solutions within the Warfighter Readiness Research Division. He enjoys tabletop gaming and video gaming, and serves as a youth group leader for his local church.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MAJOR RICHARD DILL is an Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a PhD in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; PhD 1999, Texas AM University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

III. Paper II: A Data-Oriented ECS-based Design to Send DIS Packets

The following paper, “A Data-Oriented ECS-based Design to Send DIS Packets,” was submitted and accepted by the IEEE CPS; it was published in Summer 2021 and presented at CSCE 2021.

A Data Oriented ECS-Based Design to Send DIS Packets

Noah W. Scott, Douglas D. Hodson, Richard Dill, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
emails: metaruler@gmail.com, doug@sidechannel.net, richard.dill@afit.edu,
michael.grimaila@afit.edu

Abstract— *The Entity-Component-System (ECS) is an architectural design pattern that separates data from computer logic (i.e., behavior) - components define data, systems define behavior. It is based the data oriented programming paradigm and is increasingly being used by game engines to improve execution efficiency. The Distributed Interactive Simulation (DIS) protocol is an IEEE military-oriented standard for exchanging simulation data within a distributed system to achieve real time execution performance to support human and/or hardware interaction. This effort explores implementing a interoperability interface using a Rust-based, ECS-architected simulation to send DIS packets.*

Keywords: ECS, DIS, Interoperability

1. Introduction

Training and readiness are essential in many fields, and simulations are a cost-effective medium in which to improve both of those aspects. For the military, the Distributed Interactive Simulation (DIS) protocol is the standard by which simulations can operate and function in real-time. This allows for the operators and simulation engineers to subject the trainees to a realistic training environment, which in turn improve the training experience. This effort evaluates the development of a simulation interoperability interface using a Entity-Component-System (ECS) architecture pattern in contrast to the more commonly used Object-Oriented-based approaches.

1.1 DIS

Distributed Interactive Simulation (DIS) is an IEEE standard [1] for conducting real-time platform-level wargaming across multiple host computers and is used worldwide, especially by military organizations but also by other agencies such as those involved in space exploration and medicine [2].

To facilitate *interoperability* among individual network-compatible applications (i.e., individual simulations), this standard defines the data packets (Protocol Data Units (PDUs)), and how to interpret them. This paper does not delve into all of the details associated with the standard, but focuses on one specific PDU, the Entity State PDU. Entity State PDUs are nearly ubiquitous in the sense it is understood by virtually all DIS-compliant simulations. For a good introduction to DIS, see the Simulation Interoperability Standards Organization [3] (SISO) published document titled "DIS Plain and Simple" [4].

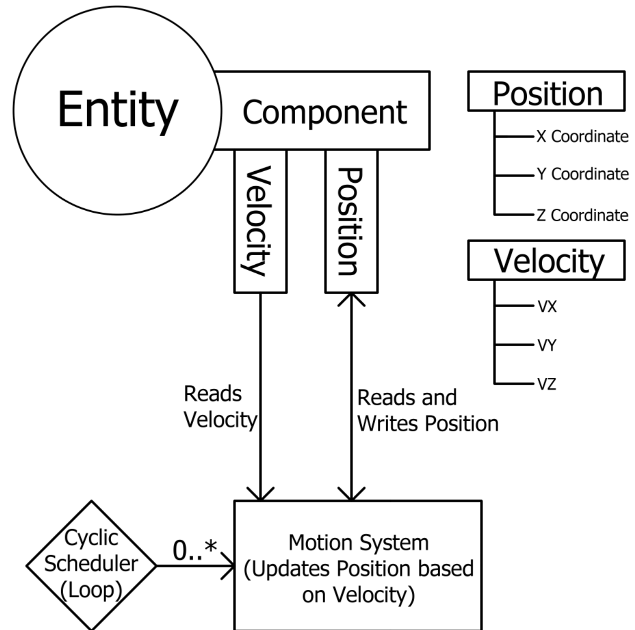


Fig. 1: Basic structure of an ECS System

1.2 ECS

ECS is an architectural pattern that organizes data and logic very differently than the more traditional Object-Oriented Programming (OOP) paradigm. Instead of defining classes and creating objects which have public methods in which to interact, we instead have entities which utilize specific components to operate either by themselves or in tandem with other entities within the system [5]. These components are the data pieces in the code, and they operate the functionality by "listening" to events when they happen. For example, if a change needs to be applied to all entities that can Walk, there is a list of each entity in the system which has the Walk component that can be utilized. This is how the system can communicate actions without having to be tied down with the inner workings of individual objects. ECS-based programming is thus designed to be more loosely created, allowing for more components to be built up without all of the hard-coded threads connecting objects together.

If another system is needed, that system can be added in by telling program which components to look for in an entity, then apply an action to each entity which has that component. An example of a single system ECS layout can be seen in Figure 1. Each time the game loop (the main loop in the program which acts as our "steps") iterates, it will run through the systems in the program that should be activated each loop. These systems are each built to perform a specific recurring action on the world, and in this case our single system is "Motion." The Motion system will check ONLY the entities in our system that have both a Position and Velocity component. This prevents each system from iterating their actions through every Entity by filtering them out based on their Component criteria. After these Entities are queried, the Motion system checks each Entity to see if they have a non-zero Velocity component. If they do, the Motion system will update the position of the Entity based on their Velocity, simulating the aspect of motion within our game. We can use this same strategy to add additional systems to our game, such as a Wind system which blows all move-able Entities in a random direction each round, or a Collide system which would check for two entities entering the same space on a loop, then apply an action based on what those colliding entities are. This allows the system to be flexible, adaptable, and easily expanded to fit the needs of the user.

1.3 HECS

HECS, more specifically, is a Rust-based Handy ECS crate (i.e., library) is designed to be a "high-performance, minimalist entity-component-system (ECS) world" [6]. The HECS crate revolves around the use of the World structure. We can place and interact with the Entities in the World in order to operate our simulation environment using the ECS architecture. Each step, the program can operate our given systems by passing the World object for our simulation to each of the systems. This allows the system to query the world for the entities it needs to interact with each step. The key basic operations built into the World object are 'spawn' and 'despawn' in addition to several methods of querying and retrieving Entity objects from the World. Each time an entity is spawned into our simulation world, we spawn it with its set of components. These are the permanent components of the entity for its entire life until it despawns, whether that be at the end of the program or the end of its life in the simulation. The query functions found within the World object such as 'query' and 'contains' can be used by our systems to perform their actions on the Entities they are set up to interact with. The basis for all systems added into the HECS environment would be that the system takes in a World by reference in addition to any other needed external variables, such as a socket for a network-based system designed to send packets. The systems within our program that send packets must all be passed a

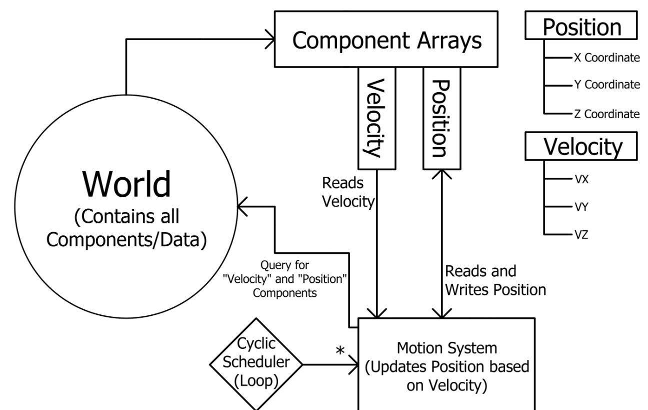


Fig. 2: A single system operation within HECS

socket, as we do not wish to open up a new network port every time we wish to send a packet across the network. Figure 2 shows our previous ECS design adapted into a HECS environment. Instead of having the game program store Entities in a list, we instead use the World within HECS to store everything we need to operate the game world. Each step in the Game Loop, our systems (in this example, the Motion System) query the World for the Components that they are looking for. Our Motion System queries for two components: Position, which indicates that an Entity exists at a specific physical location in the World, and Velocity, which indicates that an Entity can move around within the World. The World will then return all of the component pairs that the system was asking for to the Motion System.The Motion System will then read the Velocity value of each pair and apply that Velocity to the Position, adding or subtracting from each Position xyz value based on the Velocity xyz values. HECS is a robust Rust-based asset for designing our ECS system for DIS simulations, as detailed in the next section.

1.4 Software Development

Using the HECS functionality, we have created a program to show how we can bring the dynamic functionality of the ECS-based system to the DIS operational world. We have a functioning ECS system within Rust that is able to send update packets out in a similar manner to the standard DIS step functionality. Each type of PDU (such as Entity State) has a system designed to create those packets from the Entity information within the ECS. Figure **??** shows the structure of our Entity State System. This system first queries the World for the "Velocity" and "Position" components, resulting in the World returning arrays for the components which contain all of the component pairs that fit our query criteria. The Entity State System iterates through the array, reading the Velocity and Position values for each pair then creating

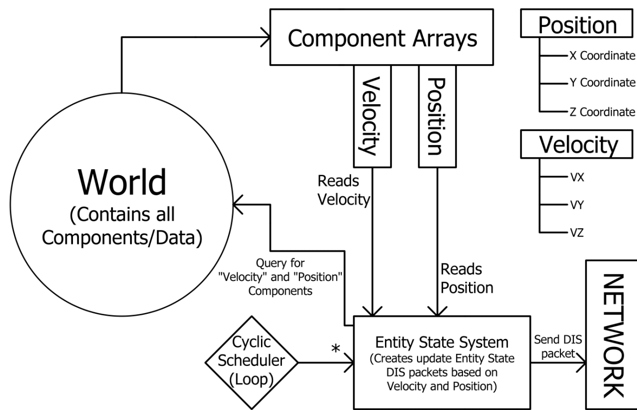


Fig. 3: Operation diagram of the Entity State System

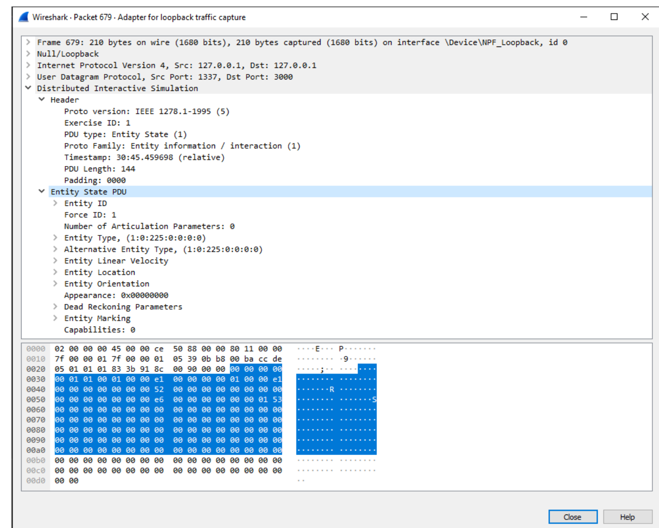


Fig. 4: Output packet sent from the program, recognized as a proper EntityState PDU

an Entity State PDU with these values plus the necessary identifiers for the "Entity". The result is a system which creates and sends a specific type of packet to the network. This individualizes the work units for each necessary packet type, and can be adjusted based on the mission being run. Our packets are recognizable through network analysis software such as Wireshark, as seen in Figure 4 [7]. The packet that we pick up here are recognizable due to their packet header, which is assembled by our program according to the IEEE standard [1]. Each part of the packet is a piece of data that is all assembled by our Entity State Packet Sending system. This system's job is to track all of the Allied and Enemy players in our World each step, then send an update packet for each of them over the network by serializing the data using Serde protocols [8]. The result is that we have a single system which we can enable and disable to send all the PDUs we need to keep track of moving entities within a simulation. The example shows one of the Allied units, identified due to the country affiliation of USA (225) [9]. Since this Entity has a magnitude to its Velocity component, each step the simulation will use the Motion system to move the Entity, then the Entity State Packet Send system to broadcast a network packet with the updated Entity position. A packet sent from this program can be read from any other simulation program as a valid DIS packet, as it utilizes a standard bincode procedure that can be deserialized on the receiving system [10]. It is designed to be within the proper specifications, so the data can be unmarshalled by a standard C++ system in order to read it properly into that simulation environment.

2. Conclusions

The work that are doing with this project is designed to evaluate designing an ECS-based interoperability interface to support distributed simulation communication. Using the HECS crate, we expect that more entities can be handled and the design of interoperability interfaces might be simplified over OOP approaches. This research will benefit the real-time distributed simulation community by understanding the pros and cons to leveraging ECS architecture over OOP-based designs. Our current software development serves as a proof-of-concept for our research with regards to this topic, and is a starting point for further development. HECS is a straightforward crate that provides all the essential functionality required to implement an ECS-based application.

3. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] "IEEE Standard for Distributed Interactive Simulation – Application Protocols," *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*, pp. 1–747, 2012.

[2] "Distributed Interactive Simulation," 2020, accessed 15-Oct-2020. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_Interactive_Simulation

[3] "Simulation Interoperability Standards Organization," 2020, accessed 15-Oct-2020. [Online]. Available: https://www.sisostds.org/

[4] "Reference for guide: DIS plain and simple," *SISO-REF-020-DRAFT*, 2007.

[5] "Entity-Component-System (ECS)," 2020, accessed 02-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Entity_component_system

[6] "Handy ECS (HECS) website," 2021, accessed 17-June-2021. [Online]. Available: https://docs.rs/hecs/0.5.2/hecs/

[7] "Wireshark website," 2020, accessed 15-Oct-2020. [Online]. Available: https://www.wireshark.org

[8] "Serde," 2020, accessed 15-Oct-2020. [Online]. Available: https://docs.serde.rs/serde

[9] "DIS data dictionary - PDU data," 2021, accessed 17-June-2021. [Online]. Available: http://faculty.nps.edu/brutzman/vrtp/mil/navy/nps/disEnumerations/JdbeHtmlFiles/pdu/d.htm
[10] "Bincode," 2020, accessed 15-Oct-2020. [Online]. Available: https://github.com/servo/bincode

Author Biographies

NOAH W. SCOTT is a 1st Lieutenant currently stationed at Wright Patterson Air Force base. He is master's student at the Air Force Institute of Technology studying software engineering concepts, primarily with regards to Simulations and Game networking. His previous work was programming within AFRL.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MAJOR RICHARD DILL is an Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a Ph.D. in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; Ph.D. 1999, Texas AM University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

IV. Paper III: Prediction and Convergence Calculations using Rust-based NAlgebra-GLM

The following paper, "Prediction and Convergence Calculations using Rust-based NAlgebra-GLM," was submitted and accepted by the IEEE CPS; it was published in December of 2021 and presented at CSCI 2021.

Prediction and Convergence Calculations using Rust-based NAlgebra GLM

Noah W. Scott, Douglas D. Hodson, Richard Dill, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
emails: metaruler@gmail.com, doug@sidechannel.net, richard.dill@afit.edu, michael.grimaila@afit.edu

Abstract— *Prediction and convergence are techniques used to reduce the network traffic between multiple distributed simulation applications that individually maintain a representation of a "world" that include moving entities. Prediction (often using dead reckoning algorithms) is an approach to estimate the position and orientation of "remote" entities hosted and/or managed by other simulation applications executing within the distributed system. Estimates are made (i.e., calculated) using previously received data, such as velocity and acceleration. As new data is received, a convergence algorithm is often used to update the "remote" entity's position and orientation within the represented world. The term "convergence" is often times referred to as "blending" or "smoothing" as its goal is to avoid visually obvious disjointed "jumps" in movement as updates are received.*

This work implements the dead reckoning estimation algorithms defined in the IEEE standard for Distributed Interactive Simulation (DIS) in software using the Rust programming language and the NAlgebra GLM package (i.e., crate/library). It also implements a simple convergence algorithm to move entities to their correct locations and orientations. This work is part of a larger design effort to prototype a DIS-compatible interoperability network interface, organized using an Entity-Component-System (ECS).

Keywords: Dead Reckoning DIS GLM Rust

1. Introduction

Simulations play a role in military training; distributed simulations allow us to connect people from different geographic locations so that the training of forces for a wide range of situations can take place without exposure to the risks involved with certain scenarios. The IEEE Distributed Interactive Simulation (DIS) standard [1] defines a protocol for how distributed simulations should interpret and exchange data to create a common understanding of a "virtual world" in which entities managed (i.e., hosted and updated) by individual simulation applications (i.e., the nodes within the distributed system) can interact. It is noted that the "virtual world" often consists of a static terrain (including water) with many static and dynamic moving entities. It is further noted that the term "virtual world" is synonymous with the terms "synthetic environment" and "synthetic battlespace."

The DIS standard defines several dead reckoning algorithms which are used to estimate (i.e., predict) "remotely" hosted entity positions and orientations so as to reduce the amount of data communicated across a network. This effort implements those algorithms in the Rust programming language using the NAlgebra GLM crate (i.e., library) as well as so-called "convergence" algorithms to smooth out, blend, or avoid disjointed "jumps" in continuous movement. This research fits within a larger effort to design a so-called "interoperability" interface within an existing Entity-Component-System (ECS) implementation.

1.1 DIS

The IEEE Distributed Interactive Simulation (DIS) standard [1] defines an approach to executing a distributed simulation using a network to exchange Protocol Data Units (PDUs). PDUs defines specific data (i.e., information) about the entities and their interactions. As an example, the entity state PDU contains information about an entity's position and orientation; it is sent to other simulation applications within the distributed system to inform them about their existence and current state.

The standard draws a clear distinction on the simulation applications role in how "locally" and "remotely" managed entities should be handled; "local" entities are the ones the simulation itself is managing (hosting and updating). State changes about "locally" managed entities are sent to the other distributed simulations using a network. When PDUs about "remote" entities are received, the standard defines the responsibilities of the simulation in terms of how data should be interpreted and processed.

Consider the the case of a missile being fired from an aircraft towards a target: The simulation managing the entity responsible for "firing" at the target entity will send a fire PDU containing the information necessary for other networked simulations to process. Processing usually means creating a representation of what was "fired" and presenting its dynamics to the user. At some point the launched (i.e., "fired") missile will detonate; when this happens the responsible simulation for determining when this happens will send a detonate PDU, and the receiving simulations

will process that information. In this case, the simulation that hosts the target will determine the outcome using the estimated position of the missile in conjunction with the "exact" or most up to date information about the target.

1.2 Prediction (Dead Reckoning)

Since the sending of entity movement by issuing entity state PDUs could consume a lot of network bandwidth, the use of prediction (i.e., dead reckoning) algorithms are employed so that "remote" entity positions and orientations can be estimated (ideally without introducing much error). Dead reckoning is a technique to estimate position and orientation; much research into how to define a better dead reckoning algorithm exists - but that is not the topic of this paper. Our interest is in the implementation of them using the Rust programming language as a set of linear algebra operations. As stated, using this technique does introduce error, but for the domain of distributed training, it is considered to be worth the price to support autonomous simulation execution which improves the responsiveness of this "real-time geographically distributed simulation."

The IEEE DIS standard (in Annex E) defines several algorithms or methods to estimate a position and orientation as shown in Figure 1; they are selected and used are based upon requirements. For example, Algorithm 3 is used to estimate the position of an object with low acceleration while also accounting for orientation; algorithm 5 estimates the position of a high speed object, while not concerned with orientation. As an example, algorithm 5 might be suitable to represent a missiles position.

Dead reckoning estimates are made for all entities within a simulation application. It is used to estimate the position and orientation for all "remote" entities, as well as to maintain estimates for "locally" managed ones. When a "locally" managed entity position and/or orientation differs enough from its dead reckoned position and/or orientation, a new entity state PDU is sent. The standard defines a default value for this difference (error); it is set based upon requirements; this bounds the error associated with using this technique.

Additionally, so-called "heartbeat" PDUs are sent at pre-determined time intervals (about 5 secs) so that late joining simulations will eventually become aware of the entities that exist.

1.3 Convergence (Smoothing/Blending)

When dead reckoning is used; how to update the position and orientation of a "remote" entity becomes an issue. Simply moving it to the newly received correct position could result in jerky movements that would be undesirable. This is the reason for so-called convergence algorithms - they more gracefully smooth or blend a dead reckoned (DR) entity's position and orientation to the correct position and orientation. This could be done in one step if the error threshold (defined for the dead reckoning algorithm) is small,

Field	Model	Formula	Examples
1	STATIC	N/A	Static entities
2	DRM (FPW)	$P = P_0 + V_0 \Delta t$	Constant velocity (or low acceleration) linear motion
3	DRM (RPW)	1) $P = P_0 + V_0 \Delta t$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 2 but where orientation is required (e.g., visual simulation)
4	DRM (RVW)	1) $P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 5 but where orientation is required (e.g., visual simulation)
5	DRM (FVW)	1) $P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2$	High speed (e.g., missile) or maneuvering at any speed
6	DRM (FPB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b)$	Similar to DRM 2 but when body-centered calculation is preferred
7	DRM (RPB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b)$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 3 but when body-centered calculation is preferred
8	DRM (RVB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b + [R2] A_b)$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 4 but when body-centered calculation is preferred
9	DRM (FVB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b + [R2] A_b)$	Similar to DRM 5 but when body-centered calculation is preferred

Fig. 1: Dead Reckoning Prediction Formulas [1]

but more likely, the position and orientation is corrected over a short time period. Even though convergence algorithms are important; they are relatively easy to implement. In many cases, they simply move an entity in steps by using interpolated points between the estimated position and correct position. Because they tend to be simple to implement, no further discussion is presented.

1.4 GLM

The OpenGL Mathematics (GLM) library facilitates graphics-oriented calculations - it is specifically designed to support the OpenGL graphics system. The GLM package defines matrices and vectors and provides linear algebra operations on them; it is implemented as a header only C++ package. It provides enough features to implement the DR algorithms presented. The Rust programming language ecosystem provides several alternative packages that define matrices and vectors and linear algebra on them.

1.5 NAlgebra

The estimation algorithms defined by the DIS standard can be implemented in software as linear algebra calculations. One such package to support this activity is NAlgebra; this package defines vector and matrix types with associated operations on them [2]. It expands on these types by adding decompositions and transformation properties such as rotation matrices and projections; many operations are available.

1.6 NAlgebra-GLM

NAlgebra is a comprehensive package, in fact it provides more functionality than we or other graphics-oriented cal-

culations require. Because of that, a subset of it has been extracted as NAlgebra-GLM.

Since computer graphic oriented calculations are a subset of all possible linear algebra operations, this subset is focused on typical "GLM" type of operations to support that domain. In other words, the NAlgebra-GLM package implements the features and capabilities that GLM provides (for C++), but for the Rust programming ecosystem. As the NAlgebra-GLM documentation states, this package exists as a "straight-to-the-point, graphics programming-oriented API" [3].

This package is particularly useful to support this research, as we are only concerned about a few dimensions, such as 3x1 vectors and 3x3 matrices. We have implemented all the dead reckoning algorithms using it. A code example for DR algorithm 4 is shown below:

```
//---------------------------------------
//DR Alg. 4 - Returns the new position &
//            orientation of the DR
//            entity using Acceleration,
//            Velocity, and Position
//---------------------------------------
fn dr_4(pos: Vec3, v: Vec3, a: Vec3,
        dt: f32, o: Mat3, drm: Mat3)
        -> (Vec3, Mat3)
{
    let new_pos = pos + v*dt + 0.5
                  * a * dt * dt;

    let new_o = drm * o;

    return (new_pos, new_o);
}
```

1.7 Software Development

To test our implementation of DIS dead reckoning (prediction) and and smoothing (convergence) algorithms, we created a ECS-based system using the Helpful ECS (HECS) [4] package. The flight dynamics model (FDM) used is a Rust implementation of the model presented here [5] - which we call the "Palmer" FDM. Using a HECS-based system from a previous implementation, we are able to store the properties the define the flight dynamics for a particular aircraft and its current state (position, velocity, etc) as two components of an entity (within an ECS). A previous implementation of the DIS interoperability interface was able to create and send DIS compliant DIS entity state PDUs without meaningful data - now it it includes dead reckoned positions and orientations.

To distinguish between "local" and "remote" entities, we created a single component that only contains a Boolean value to serve as an identifier; this is used to filter (i.e., query) for the entities of interest contained by the ECS.

Additional information is stored as a component that indicates other aspects of interest, such as which dead reckoning algorithm is used. Sending this information along with the PDU ensure all simulations are in sync in terms of how prediction estimates are to be calculated.

Our dead reckoning system is presented as follows:

```
fn system_dead_reckon_entities(world:
   &mut World, dt: f32)
   {
    // Perform component checks for each
    // DR entity in the system
    for (_id, (_prop, mut state, d)) in
    &mut world.query::<(&Properties,
                         &mut State,
                         &DeadReckonData
                         )>()
    {
        let mut vec_pos =
            nalgebra_glm::vec3
            (
            state.q[1], state.q[3],
            state.q[5]
            );
        let vec_v =
            nalgebra_glm::vec3
            (
            state.q[0], state.q[2],
            state.q[4]
            );
        let vec_a =
            nalgebra_glm::vec3
            (
            state.fx, state.fy,
            state.fz
            );

        let phi = state.q[0]
            .atan2(state.q[2]);

        // Angle conversions for
        // orientation
        let sin_psi =
           state.bank.sin();
        let cos_psi =
           state.bank.cos();
        let sin_theta =
           state.alpha.sin();
        let cos_theta =
           state.alpha.cos();
        let sin_phi =
           phi.sin();
        let cos_phi =
```

```
            phi.cos();

        let a11 = cos_theta * cos_phi;
        let a12 = cos_theta * sin_phi;
        let a13 = sin_theta * -1.0;
        let a21 = (sin_psi * sin_theta
                  * cos_phi) -
                  (cos_psi * sin_phi);
        let a22 = (sin_psi * sin_theta
                  * sin_phi) +
                  (cos_psi * cos_phi);
        let a23 = sin_psi * cos_theta;
        let a31 = (cos_psi * sin_theta
                   * cos_phi) +
                   (sin_psi * sin_phi);
        let a32 = (cos_psi * sin_theta
                  * sin_phi) -
                  (sin_psi * cos_phi);
        let a33 = cos_psi * cos_theta;

        // Angular Velocities for
        // orientation
        let mut mat_o =
            nalgebra_glm::mat3(a11, a12,
                a13, a21, a22, a23, a31,
                a32, a33);

        let mut result =
            (vec_pos, mat_o);

        match d.algorithm
        {
        // Match based on required
        // algorithm
        ...
            4 => result = dead_reckon_4(
                 vec_pos, vec_v, vec_a,
                 dt, mat_o, drm),
        ...
            _ => println!("No DR"),
        }

        vec_pos = result.0;
        mat_o = result.1;

        state.q[1] = vec_pos[0];
        state.q[3] = vec_pos[1];
        state.q[5] = vec_pos[2];
    }
}
```

We used Wireshark [6] to capture and analyze the PDUs sent; initial results show PDU generation with correct DR information as expected. When the dynamics of our entity using the "Palmer" FDM, we selected algorithm 4 to represent its position and orientation, as it seems to be the most appropriate.

2. Conclusions

This effort's purpose was to continue the development of our Rust-based, ECS-based, DIS interoperability interface. This step addresses the implementation and application of both prediction through dead reckoning and convergence through a simple interpolated stepping from estimate to true value.

The algorithms were implemented using a linear algebra approach using the Rust-based NAlgebra-GLM package. Testing to ensure it works as intended was accomplished using Wireshark to dissect DIS PDUs. The next step in this research direction (a software engineering design effort to implement a ECS-based, DIS-focused interoperability interface) is to add systems to create "remote" entity representations when received.

3. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] "IEEE Standard for Distributed Interactive Simulation – Application Protocols," *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*, pp. 1–747, 2012.

[2] "nalgebra," 2021, accessed 30-July-2021. [Online]. Available: https://nalgebra.org/

[3] "nalgebra-glm," 2021, accessed 31-July-2021. [Online]. Available: https://docs.rs/nalgebra-glm/0.15.0/nalgebra-glm/

[4] "Rust HECS crate site," 2021, accessed 17-June-2021. [Online]. Available: https://docs.rs/hecs/0.5.2/hecs/

[5] G. Palmer, *Physics for Game Programmers*. USA: Apress, 2005.

[6] "Wireshark," 2021, accessed 6-Sep-2021. [Online]. Available: https://www.wireshark.org/

[7] "Entity-Component-System (ECS)," 2020, accessed 02-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Entity_component_system

[8] "Distributed Interactive Simulation," 2020, accessed 15-Oct-2020. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_Interactive_Simulation

Author Biographies

NOAH W. SCOTT is a 1st Lieutenant currently stationed at Wright Patterson Air Force base. He is master's student at the Air Force Institute of Technology studying software engineering concepts, primarily with regards to Simulations and Game networking. His previous work was programming within AFRL.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and

both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MAJOR RICHARD DILL is an Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a Ph.D. in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; Ph.D. 1999, Texas AM University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

V. Paper IV: An ECS-Based DIS Interoperability Interface

The following paper, "An ECS-Based DIS Interoperability Interface," is a submission that is to be determined.

An ECS-based DIS Interoperability Interface

Noah W. Scott, Douglas D. Hodson, Richard Dill, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
emails: metaruler@gmail.com, doug@sidechannel.net, richard.dill@afit.edu,
michael.grimaila@afit.edu

Abstract—*An Entity-Component System (ECS) is a software architecture where the components (i.e., data) define the entities are stored as organized lists which are operated upon by systems (i.e., behavior). The IEEE Distributed Interactive Simulation is a standard that defines a set of protocol data units (PDUs) that are exchanged networked simulations within a distributed system. The standard also defines the semantics for how each simulation is suppose to send, receive and process the PDUs. The standard is often used by military simulations to interconnect differently developed software applications to each other, either at the same or significantly different geographic locations.*

When leveraging a network to exchange PDUs, each simulation can represent a common understanding of the world at the desired level of detail and allow hosted entity interactions. Often, DIS compliant simulations are written in an Object-Oriented Programming paradigm, where data is encapsulated into objects and state is changed via exchanged messages (i.e., method calls) through a publicly defined interface. This research investigates an ECS-compliant IEEE DIS interoperability interface by developing a prototype application that can send, receive, and process Entity State PDUs as defined by the standard.

The prototype is written in Rust programming language. It depends upon the nalgebra-glm library (i.e., crate) to provide general-purpose linear algebra features to perform the mathematical operations associated with prediction (i.e., dead reckoning) and convergence. It also depends upon Serde to perform the required serialization and deserializtion of PDUs and finally, it depends upon the Handy ECS (hecs) library to provide a minimalist entity-component-system (ECS) capability. Our prototype implements a simple cyclic scheduler to execute the ECS systems (i.e., behavior) that update the components.

Keywords: ECS IEEE DIS Rust Distributed Simulation

1. Introduction

Simulation technology is prominent in many different sectors of the world, but we are specifically focusing on simulations with application to training pilots. Using simulations, one can provide a wide range of possible situations from the most simple of tasks to a full-scale flight training scenario. A significant advantage to this approach is the cost savings versus the possibilities that the simulations can bring about for any given group in need of a specific proficiency. In addition, when it comes to military environments a simulated training world can alleviate the possible risks involved in a live-fire exercise while still maintaining the possible threat of a “real” battle.

This research concerns the creation of a prototype DIS interoperability interface that can send, receive and process Entity-State PDUs as defined by the IEEE standard. The interface is organized as an Entity-Component System (ECS). The software concerns the creation of entities that are locally managed in addition to the creation and handling of remote entities that are managed by other devices within a distributed simulation. Previous publications pertaining to this topic include the creation of IEEE DIS Entity State Protocol Data Units (Entity State PDUs), i.e. binary packets to send data structures [1], a Rust-based ECS designed to send the packets [2], and finally the algorithms needed to perform both prediction and convergence during the simulation [3].

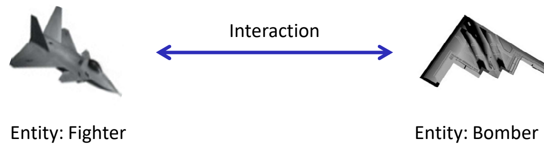
2. Background

2.1 Distributed Interactive Simulation

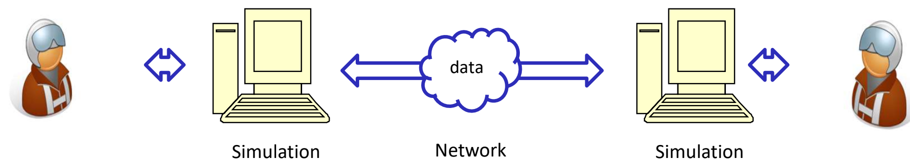
A simulation can be built in two distinct ways from a hardware perspective. The first way would be a simulation which uses a single system to handle all operations and interactions. Each *player* would connect their controls to the system and operate from that single location. This would be similar to a stand-alone aircraft crew operating a single plane which did not encounter any external interactions. The other (and more common) way of building simulations is by connecting multiple systems together through either Local Area Connection or a Networked Connection, i.e. a peer-to-peer design approach. This effort focuses on Networked (Distributed) Interactive Simulations that exchange data with other peers. The simulations themselves are often executed at different geographic locations and exchange the defined PDUs using a network in order to maintain a mostly consistent view of the world in which hosted entities interact.

Figure 1 shows the three views (or perspectives) of the Distributed Interactive Simulation system. Within the top level (Operational), the aircraft entities move within the virtual world, and will take certain actions that affect either a piece of the world environment or another entity. At the System level, the *player* is able to see their own aircraft’s

Operational View



System View



Data Consistency View

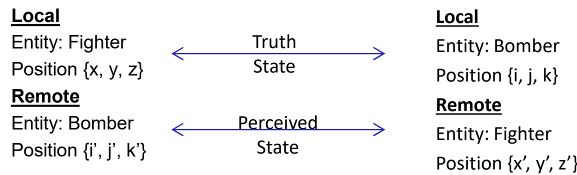


Fig. 1: Three Views of a Distributed Interactive Simulation

status such as altitude and bank angle, and can take actions or reactions to the information they receive through the network. This network data is translated into a more intuitive form for the *player* (such as a lock-on signal or a missile launch), but beneath the System level is where the software-level data is handled. The data consistency view shows that each system contains one of the two aircraft in the simulation, labelled as a *local* entity. However, each system must be aware of the other entities, so they store all non-local entity data as *remote* entities. The individual connected systems also handle inputs and outputs for only the *local* entities in the Distributed Interactive Simulation environment through the user controls (inputs), received simulation packets (inputs), and sending network packets (outputs). All of the remote entities in a given system are instead extrapolated using an equation of motion to predict the movement of each entity. This method allows for the simulation to appear smooth while also reducing total network usage, as less update packets will be needed for each entity.

To explain DIS operation further, we can walk through an example interaction between two different hosted entities. Consider an aircraft launching a missile at a ground target such as a bunker. Since this is a distributed system, each simulation will control specific entities. Simulation A locally manages the aircraft and the missile loaded onto the aircraft as *local* entities, and Simulation B locally manages the ground bunker as its *local* entity. As the aircraft flies in range of the bunker, it fires the missile towards it, causing Simulation A to send a packet to the network. This packet tells all other simulations on the network that aircraft A has launched missile A, and this triggers the creation of *remote* missile entities within other simulations in which they will update position as needed. While the missile is in flight, simulation B detects that the missile has overlapped the

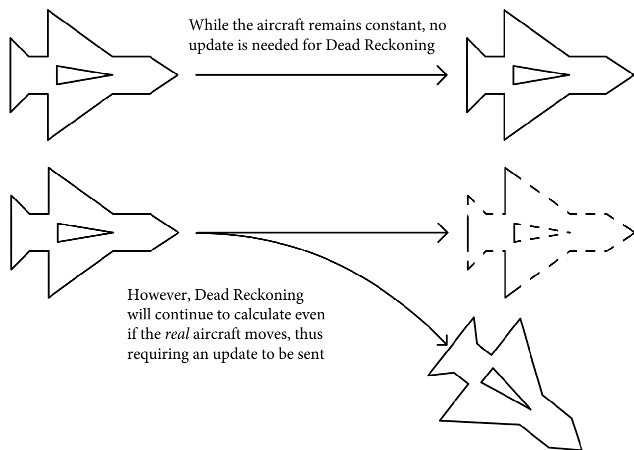


Fig. 2: An example of when update packets are needed

location of the bunker that it locally manages, which will now trigger another packet to be sent. This packet contains the location and results of the missile impact. Simulation B uses this result to apply the damage to the bunker that it locally manages, then sends an update packet containing the updated information for the now-damaged bunker. The packet allows all other simulations on the network to update their *remote* versions of the building to reflect this missile strike, keeping the distributed system mostly consistent. Figure 3 is a graphical summary of this interaction.

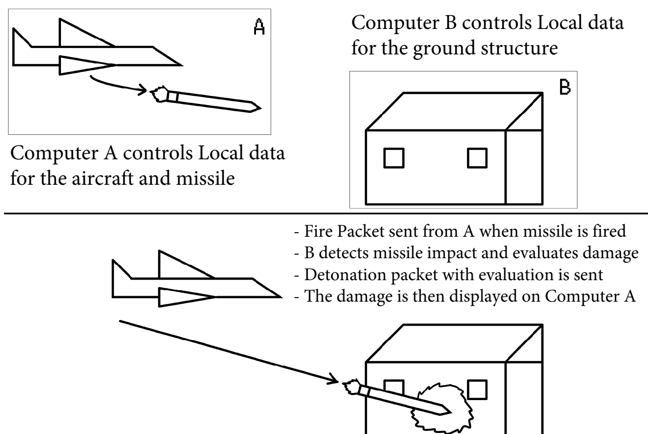


Fig. 3: Example two simulation entity interaction within a DIS-based system

2.2 Prediction And Convergence

If all updates within a given simulation required a packet to be sent, the network on which the simulation ran would require an exponentially large bandwidth to accommodate real-time requirements. In order to prevent such a situation, Distributed Interactive Simulations rely on a process known as Dead Reckoning to perform routine motion calculations locally.

Dead Reckoning is performed on each *remote* entity until a packet corresponding to that entity is received over the network. At this point, the *remote* entities are likely “out of sync” with the new “current” data received from the machine holding the *local* data (an example is shown in Figure 2). As a result, the new packet becomes the base for Dead Reckoning, but a straight jump between the two locations will likely result in non-realistic movement and jumps within the simulation. To remedy this problem, we have also implemented a Convergence technique. Each time a *remote* entity must be updated, the program checks the distance between the current Dead Reckoned position of the entity and the “Real” position received over the network. If the distance exceeds a realistic threshold of movement between two simulation steps, the Convergence technique will instead move the entity a maximum distance based on the factors determined by the Dead Reckoning algorithm (velocity, acceleration, and/or orientation) each step until that data has converged to the proper location, from which it will continue to extrapolate its position using its assigned Dead Reckoning algorithm. By doing this, overall movements within the simulation are smoothed out without needing to shrink the synchronization threshold for sending update packets.

Requirements for a DIS IEEE system
No central server/computer; each node has their own projection of the world
Each node must control at least 1 simulation entity
Nodes must be able to communicate "truth data" for their local entities over the network using PDUs
Changes to an entity are communicated from the node that controls it locally
Event and remote entity perception is determined by the node receiving the data
Dead Reckoning calculations are utilized to reduce network traffic by predicting remote movement

Table 1: DIS IEEE Requirements

2.3 DIS IEEE Standard

Standards are written in order to enforce a specific network-based simulation design operation. We utilize the Distributed Interactive Simulation standard (DIS), which is intended to conduct real-time interactions over multiple machines [4]. The standard has been used in an ever-expanding collection of simulations from military exercises to medical practice, and is the most applicable baseline for our line of effort. This standard is used primarily within Air Force based distributed simulation applications, which is a

major motivation behind building this software around flight applications.

Field	Model	Formula	Examples
1	STATIC	N/A	Static entities
2	DRM (FPW)	$P = P_0 + V_0 \Delta t$	Constant velocity (or low acceleration) linear motion
3	DRM (RPW)	1) $P = P_0 + V_0 \Delta t$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 2 but where orientation is required (e.g., visual simulation)
4	DRM (RVW)	1) $P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 5 but where orientation is required (e.g., visual simulation)
5	DRM (FVW)	1) $P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2$	High speed (e.g., missile) or maneuvering at any speed
6	DRM (FPB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b)$	Similar to DRM 2 but when body-centered calculation is preferred
7	DRM (RPB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b)$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 3 but when body-centered calculation is preferred
8	DRM (RVB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b + [R2] A_b)$ 2) $[R]_{w \to b} = [DR][R_0]_{w \to b}$	Similar to DRM 4 but when body-centered calculation is preferred
9	DRM (FVB)	1) $P = P_0 + [R_0]^{-1}_{w \to b}([R1] V_b + [R2] A_b)$	Similar to DRM 5 but when body-centered calculation is preferred

Fig. 4: IEEE Dead Reckoning Prediction Algorithms [4]

A major component of the IEEE Standard is sending and receiving standard-compliant packets known as Protocol Data Units (PDUs). These packets define network data for simulation interaction, such as the data for a missile launch or necessary information to simulate an aircraft at a given time. For this software, we focused on the Entity State PDU type. Entity State packets contain critical information about any given object (or entity) within the simulation. These PDUs are processed by the machines connected to the simulation in order to determine not only the location of each object in the simulation, but it also carries information regarding physical attributes such as velocity and acceleration of that object. With this information, the simulation program is able to communicate to the connected machines to update *remote* data so that each player on these machines can see simulated effects in real time, allowing the standard to fulfil the goals of a Distributed Interactive Simulation listed in Table 1.

Regarding Prediction techniques, Annex E of the IEEE DIS Standard contains a table of 9 specific Algorithms to be used when performing Dead Reckoning calculations (as seen in Figure 4). Each Algorithm is used by certain types of Entities to fulfil prediction requirements. If the entity is a high-speed object, for example, it is likely that Algorithm 5 will be associated with it, as it deals with those types of entities where orientation is not as important as tracking motion. This algorithm would not be as suitable for an aircraft, however, as the orientation can be critical in determining simulation factors and effects, as well as

Requirements for an EntityState system
Translate stored local World entity data into PDU format for sending
Translate received EntityState PDUs into local World entity data
Monitor whether received PDUs are updates to a previously received entity OR a new one to be added
Apply movement calculations and user-controlled inputs to all locally held entities

Table 2: EntityState System Requirements

displaying the graphic properly within the simulation view. As such, Algorithm 3 would be a more likely candidate for this type of Dead Reckoning. A more in-depth analysis and implementation of these Algorithms can be found in our previous work [3].

2.4 Serialization/Deserialization

To fulfil the standard of sending and receiving PDUs within a DIS simulation, the software needs a method of "Serializing" local data into properly formatted packets to send over the network. For this effort, we've chosen to use the Serde "Crate" (Rust's version of an import package in C++). Serde is "a framework for serializing and deserializing Rust data structures efficiently and generically" [5]. The process of "Serialization" is to take a collection of data (in this case our assembled PDU packet) from Rust and transform it into a network-ready form that can be sent over a standard port. Once the recipient (or recipients) receive this data, they can then use "Deserialization" to transform the network data back into the data components from which it was assembled, completing the proper transfer of simulation data in accordance with the DIS standard.

The Serde Crate is capable of not only serializing and deserializing itself, but is also able to operate upon a wide range of other data types (including the ones used specifically in this effort). Since Serde's systems are already built to handle all of our needed use-cases, it's been included as the software's method of network packet interaction.

2.5 ECS

ECS is "an architectural pattern that organizes data and logic differently than the more traditional Object-Oriented Programming (OOP) paradigm" [2]. This style of coding defines the "objects" of the program as entities of data which are operated on using built systems for each desired functionality [6]. Entities within this system are essentially multiple specific data types linked together to represent a single unit within the World. For example, if one wanted to store a simple Car within a given space, they could store an Entity which contains Speed, Direction, and FuelRemaining components. The program could then use those components to identify that this Entity should be operated upon by a UseFuel system each step that the Car is running. The reason such a system would know to operate on the Car is because

the Car Entity has a Gas Remaining component which the system would check for. To expand upon this example, let's assume that the user then adds a Truck and Airplane vehicle to the space. Both of these Entities would also have a FuelRemaining component, as they run on fuel. As a result, the UseFuel system could also be applied to these new Entities, as it is only dependent on any given Entity having a FuelRemaining component to be operated upon. Each piece of data is treated as a different "storage type", which is what allows for a query to check for the presence of these types. Unlike objects, this data is decoupled from the other pieces of data it's associated with, allowing for the systems to operate upon each storage type as needed without having to pull the entire entity. This example shows one advantage of the ECS system, as two different Entities within a game space can utilize the same coded system during a simulation to reduce the need for specific iterations of functions.

If one wanted to increase the complexity and variability of the simulation, they could also add a FuelConsumption component to each of the Entities, which identifies how much fuel each Entity uses on each step. Then, the UseFuel system could check for both components and use the FuelConsumption component to decrement the FuelRemaining each time, which means the three entities with those components could have 3 different outcomes based on their own individual data while still using a single system for operation. A new system can be added for any operation required within the program, such as a Steering system which checks for Entities with Direction and SteeringWheelPosition components and affects the Direction the vehicle is facing each step based on the current Direction adjusted based on the SteeringWheelPosition. The ECS system is a modular design which can be adapted to any level of complexity required by the programmer and user.

In order to illustrate these techniques, let's use an example. Let's assume that, in a given simulation, there is an aircraft travelling at a constant speed in a northward direction. This aircraft also maintains constant altitude. As long as this aircraft remains at this constant cruising speed and altitude, the only update PDUs that will be sent are Heartbeat PDUs at predetermined times, indicating that no significant changes have occurred to warrant a Dead Reckoning image update. Now, assume this plane begins to bank to the right at a given time. Once the plane begins to bank, its flight direction will obviously change. However, all of the systems calculating this entity remotely will still be operating under the assumption that the plane is flying straight north. Therefore, once this plane breaches the threshold for divergence, it will send an update packet to the network. This packet will indicate that the plane is banking to the right at a given rotational rate and is beginning to shift direction as a result. At this point, all of the *remote* simulation machines will update their images of this given aircraft to match this new data, which will assume that the plane is now rotating to the right

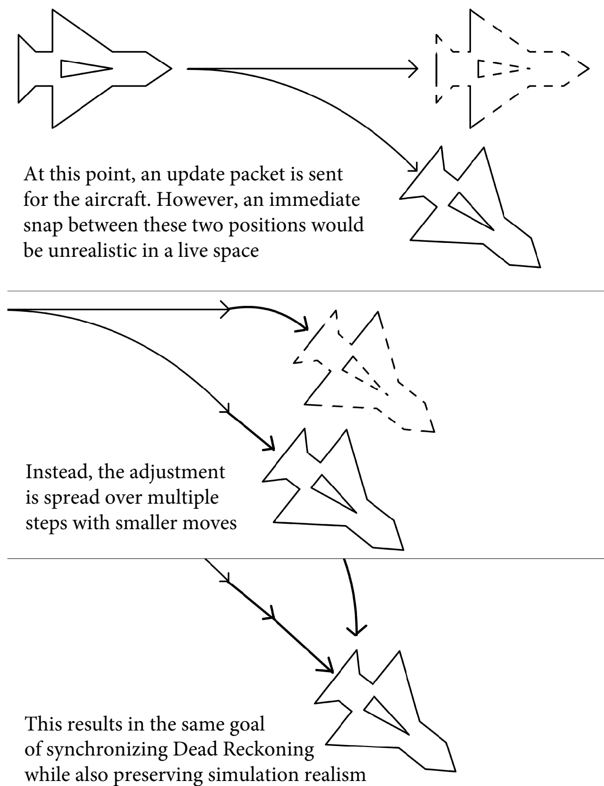


Fig. 5: An example of smoothing Convergence

at a constant rate. Once the plane reaches its intended bank angle, it will then stop rotating, which will trigger another threshold breach. This is because where the *real* plane has stopped rotating, the *remote* images of the plane will still extrapolate the rotation further until updated. As a result, another update Entity State PDU is sent in order to reflect this rotational halt to the *remote* systems. The plane now stays in this rotational angel until it reaches the point where it begins to rotate left to level out. During this time, no packets asside from Heartbeats are needed as the speed, change in direction, and bank angle have remained constant. However, as the plane begins to rotate left, a new packet is sent to once again update the network, which will stay constant until leveling out and sending another. At this point, the aircraft is flying directly East with a constant speed and altitude, which warrants only Heartbeat packets.

If we disregard the regular Heartbeats, an aircraft turning from North to East would only require 5 packets to fully complete the maneuver, with the rest of the time being extrapolated by Dead Reckoning: Fly North, rotate Right, stop rotate, rotate Left, stop rotate. If not for this system, a maneuver such as this would cost packets based on the sending rate of the simulation, which would be significantly higher cost to the network than the local calculations en-

forced within this system.

3. Software Design

As stated previously, this effort was built as an ECS interface to be used in Distributed Simulation applications. All of the data for each entity (in this case the aircraft) is contained within 3 specific components: Properties, State, and either an EntityInfo or DeadReckoningData storage component. The Properties component contains all of the specific static data for the aircraft such as wing span, engine power, and mass. In contrast to Properties, the State component contains all of the dynamic variables for the aircraft. The Time variable is automatically updated as the simulation progresses, and the positional array contains six floating-point variables which will change depending on the inputs to Bank (roll), Alpha (pitch), Throttle, and Flap Deflection. These four quantities are affected by user inputs, and are controlled throughout the simulation to allow the user to actually "fly" the aircraft in the simulated space.

The third component will differ between Entities, depending on whether the Entity is a *local* or *remote* object. The EntityInfo component contains a Boolean to indicate whether the object is being locally controlled. The DeadReckonData component instead contains variables used in the calculation and application of the program's Dead Reckoning and Smoothing functions. The first of these variables is Algorithm, a numerical value that indicates which of the algorithms found in Figure 4 are used when calculating the Dead Reckoning position of the containing Entity. The other two variables HeartbeatCounter and Syncing are used to indicate whether the Entity needs to send a Heartbeat packet and whether or not the containing Entity is currently re-syncing to the proper position of its *local* data counterpart. Each *remote* Entity is also linked with the *local* ID of its represented Entity for synchronization and removal purposes, which will be addressed in the next section.

The software can operate both as a Flight Sim and a Packet Receiver. While running, the software will take user input to fly the aircraft while sending packets in accordance with the Distributed Interactive Simulation space. At the same time, the software will receive and track EntityState packets that are sent over the network, adding this EntityState data to the local image as *remote* Entities or updating pre-existing entities with new packet data. The simulation node functions in accordance with the specifications detailed in Table 2.

While in Flight Sim mode, the software first creates the aircraft Entity by using preset values for Properties, State, and EntityInfo. This Entity is created within the World construct, where it can be queried and updated as needed. Another Entity is then created using the same Properties and State, but will instead include a DeadReckonData and EntityID component to identify it as a *remote* entity image. This allows the *local* controller to see where the remote systems project this *local* entity's status and location using the appropriate Dead Reckoning algorithm, which allows for the system to know when a synchronization packet must be sent. For networking, the software then binds to a preselected UDP Socket to send packets over the network. The final step in the setup is to add variables which will track keyboard inputs that the user will press to control the aircraft State variables. At this point, the Flight Sim enters its standard Game Loop: check for keyboard input, update aircraft position and velocity based on current variables, perform Dead Reckoning calculations, then send packets if needed. This loop will run until the user inputs the quit command by pressing the Q key. All of these inputs can be found listed in Table 3.

Key Input	Simulation Action
E	Increase Throttle
D	Decrease Throttle
Up Arrow	Increase Alpha (Pitch)
Down Arrow	Decrease Alpha (Pitch)
Left Arrow	Rotate (Roll) Plane Left
Right Arrow	Rotate (Roll) Plane Right
L	Aim Wing Flaps Down
K	Aim Wing Flaps Up
Q	Exit Simulation

Table 3: Keybinds For Software Operation

Each of the loop steps employ systems we have built to perform the needed calculations. These systems are Flight IO, Dead Reckon Entities, and Send Entity State. Each system description is accompanied by a graphic flowchart of their process.

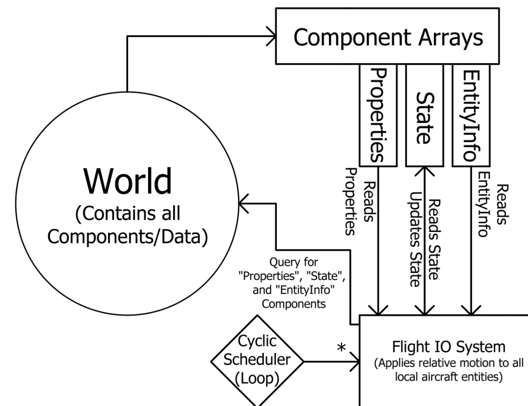


Fig. 6: Operation diagram of the Flight IO System

Flight IO (Figure 6) handles the input and output of the aircraft, taking in the current keyboard inputs and the World and returning the current key press to be checked against in the next loop. Within this function, the World is queried for all local aircraft entities (we use only one for our specific application), then applies the Equations Of

Movement to each aircraft to move them forward one "step" in the simulation space based on their current variable values. The State of this Entity is then printed to the screen to reflect the changes that the user applied through their input in addition to the movement calculation.

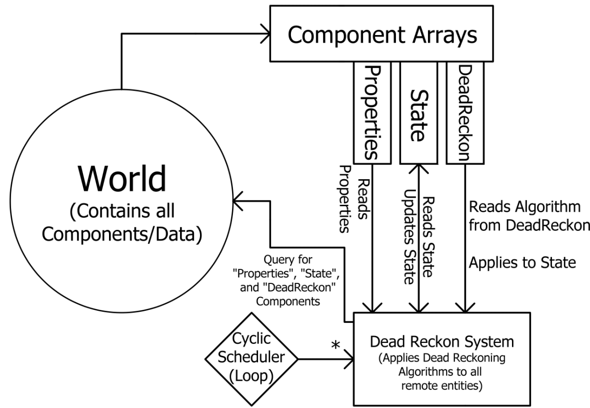


Fig. 7: Operation diagram of the Dead Reckon Entities System

Dead Reckon Entities (Figure 7) occurs right after moving the *local* entities, and by contrast queries the World for any *remote* entities to apply their respective Dead Reckoning Algorithm4 for the given step. First, the system calculates several required values for the algorithm using the State and Properties of the Entity, such as the Position Vector, Velocity Vector, and Orientation Matrix. The function then checks the DeadReckonData property of the Entity to select which Algorithm will be used to calculate the Dead Reckoning result. The results of this calculation are then stored back to the Entity in order to update their information within the simulated area and to any visuals that the simulation is linked to. These steps occur for each *remote* entity present within the World.

Send Entity State (Figure 8) is a system that handles checking for Dead Reckoning synchronization and sending packets if needed. The reason this sync is checked here is that any mismatch exceeding a given threshold between *local* and *remote* data for a given entity will require a packet be sent to the network. This function is also responsible for sending the Heartbeat packet when needed. To check proper synchronization, the system first queries the World for a *local* entity, then uses the Entity ID from this Entity to retrieve the Dead Reckoning version of the Entity from the World using a direct "get" function. The function then runs through all of the positional data of both entities, taking the difference between each one and checking that difference against the set threshold. If this threshold is exceeded by one or more parameters, the Desync flag is set. After checking the positional difference, if Desync is

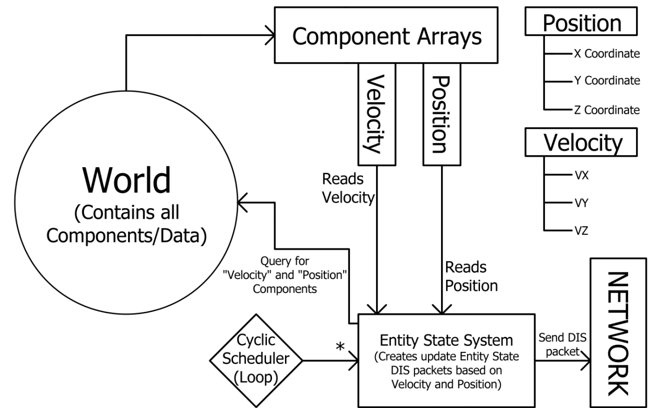


Fig. 8: Operation diagram of the Entity State System

not set then the function ends for that Entity. Otherwise, the process of Convergence5 is initiated by setting the Syncing flag within the DeadReckonData Component of the Entity, which will indicate to the function over the next two steps that the Entity is realigning with its *local* counterpart, or "Converging" [3]. After the Dead Reckoning Data matches the Local data once again, the Entity exits Syncing mode and continues functioning as a standard Dead Reckoning calculation until either a Heartbeat or Desync occurs. The Heartbeat counter is reset whenever a packet is sent as well, so any Convergence steps taken will also reset this counter. A packet is only sent from this system when either a Convergence is required or the Heartbeat counter reaches the predetermined time interval.

Upon exiting the Send Entity State System, the thread is put to sleep for a set amount of time (in our case, we calculate this time using the inverse of the Frame Rate multiplied by 100 ms), then the next step begins at the start of the loop. As stated previously, this loop will continue to run until manually stopped by the user. All of these systems and their application to the DIS Standard can be found in Table 4.

4. Implementation

4.1 Rust

Rust is a programming language which functions as a data-oriented language in contrast to Object-Oriented methods. Additionally, the Rust compiler will not allow for "bad" coding practices when it comes to data ownership. For each function, the programmer is required to specify whether or not each piece of data is being passed by reference or by value. This means the function either gets a pointer to the "real object" (meaning any changes will reflect on the object even after function completion), or the value is simply copied to be used by the function. However, once an object's

System Name	Function	Standard Application
FlightIO	Applies Movement to Locally Controlled Entities	Each node must control at least 1 simulation entity
DeadReckoning	Use Dead Reckoning prediction to reduce net traffic	Dead Reckoning calculations are utilized to reduce network traffic by predicting remote movement
EntityState	Handles assembling EntityState packets	Changes to an entity are communicated from the node that controls it locally
Sender	Sends packets to network from queue	Nodes must be able to communicate "truth data" for their local entities over the network using PDUs
Receiver	Recieves packets from the network and adds them to the local world	Event and remote entity perception is determined by the node receiving the data

Table 4: Software System Layout

ownership is passed out of a programming scope, such as from Main to a sub-function, it will flag that object if the programmer attempts to use it again to identify that it may trigger a data dependency.

Another main reason why Rust is a prevalent alternative is that it does not utilize a “Garbage Collector”. In C++ (as well as many other programming languages), the Garbage Collector is a piece of the language back-end which performs inherent Memory Management. It will look for memory that has been de-allocated by the program and return the memory to the system automatically. The problem with such a seemingly convenient construct is that it also requires additional processing time on top of the program’s original requirements. This means that if we have two copies of a program and only one uses this technology, the one without the Garbage Collector will run faster. The trade-off from not having this is that the user is required to ensure their own memory allocation structure in order to prevent memory issues such as Memory Leaks, where the program allocates memory but does not de-allocate it before the end of the program execution. The Rust compiler alerts the programmer of these things as well. Rust will not compile if there’s a chance of memory allocation issues, and will prompt the user to fix those issues before they can create the running program file. The results are a stricter system which operates with a lower average run time thanks to the absence of unneeded automatic pieces.

4.2 Serde

Because Serde is able to serialize and deserialize data, we have integrated it into the software as part of the Sender and Receiver systems. The Sender system uses Serde to transform assembled PDU packets into bincode (or Binary Code), which is a string of binary that can be easily sent over the network to the other simulation nodes. On the receiving end, the Receiver system uses Serde to turn this string of binary back into a PDU packet and perform its other actions to check how it affects the world. It is due to Serde’s versatility that we are able to turn a user-made struct like the PDU packet into the built-in bincode type.

4.3 HECS

The Handy Entity-Component System (HECS) Library is an implementation of the ECS standards discussed above to the Rust environment. This crate is designed to be a simple-to-understand package that comes with structures such as World that can hold entity data. It also provides a Query method that may be used to retrieve stored World data based on the criteria required by any given System within the code. Using HECS, our previous ECS example could be implemented as shown in Listing 1.

Listing 1: A HECS Example for vehicles

```
//--------------------------------------
// An ECS system containing a Car, Truck,
// and Plane, implemented using HECS
//--------------------------------------
use hecs::*;
use components::*; //Our components file
// The Components file contains the types
// Speed, Direction, and FuelRemaining,
// which are all user-defined types
// containing a single numerical type.
// Each entity created within this World
// will contain all 3 of these components

fn main()
{
    let mut world = World::new();

    let car_speed = Speed::new(45);
    let truck_speed = Speed::new(35);
    let plane_speed = Speed::new(200);

    let car_direct = Direction::new(0);
    let truck_direct = Direction::new(90);
    let plane_direct = Direction::new(180);

    let car_fuel = FuelRemaining::new(25);
    let truck_fuel = FuelRemaining::new(47);
    let plane_fuel = FuelRemaining::new(60);

    // Spawn Entities with given components
    world.spawn((car_speed, car_direct,
                 car_fuel));
    world.spawn((truck_speed, truck_direct,
                 truck_fuel));
    world.spawn((plane_speed, plane_direct,
                 plane_fuel));
    .
    .
```

```
    .
}
```

We utilize the HECS implementation in order to streamline the process of setting up the ECS environment in addition to the pre-built functionality that HECS provides. For example, the EntityState system which sends update packets for all *local* entities uses the Query function provides in HECS to access only the entities in the world that are local to the system. This filters out entities such as Dead Reckoning Data and *remote* entities from other systems which are calculated using the Dead Reckoning algorithms as opposed to the local controls. Entities that must be removed (i.e. a missile that has already detonated, a remote entity that has been disconnected) are also able to be removed using pre-built functions within the HECS system. Any future systems built upon this effort can be integrated by following the ECS common component structure, as any collection of components can be added to the World to create an entity within HECS. The systems and requirements they fulfil for IEEE DIS can be found below:

4.4 Code Reference

Listing 2: Properties and State Components

```
fn gen_prop() -> Properties {
    let prop = Properties {
        wing_area: 16.2,
        wing_span: 10.9,
        tail_area: 2.0,
        // slope of Cl-alpha curve
        cl_slope0: 0.0889,
        // intercept of Cl-alpha curve
        cl0: 0.178,
        // post-stall slope of Cl-alpha curve
        cl_slope1: -0.1,
        // post-stall intercept of
        // Cl-alpha curve
        cl1: 3.2,
        // alpha when Cl=Clmax
        alpha_cl_max: 16.0,
        // parasite drag coefficient
        cdp: 0.034,
        // induced drag efficiency
        // coefficient
        eff: 0.77,
        mass: 1114.0,
        engine_power: 119310.0,
        // revolutions per second
        engine_rps: 40.0,
        prop_diameter: 1.905,
        //  propeller efficiency coefficient
        a: 1.83,
        //  propeller efficiency coefficient
        b: -1.32,
    };
    return prop;
}

fn gen_state() -> State {
    let state = State {
        time: 0.0,          // time
        q: [0.0; 6],        // ODE results
        bank: 0.0,          // roll angle
        alpha: 4.0,         // pitch angle
        throttle: 0.0,      // throttle percent
        flap: 0.0,          // flap deflection
    };
    return state;
}
```

Listing 3: Local and Remote Indicator Components

```
#[derive(Debug, Default)]
pub struct EntityInfo {
  pub local: bool,
}

#[derive(Debug, Default)]
pub struct DeadReckonData {
  pub algorithm: u8,
  pub heartbeat_counter: u8,
  pub syncing: bool,
}
```

5. Conclusions

The purpose of this effort was to create a basic DIS Interoperability Interface built as an Entity-Component System. The resulting software is now able to process both its own *local* entities as well as any number of received *remote* entities using the appropriate techniques. An intentional benefit to the ECS organization of this effort is that the Systems and Components are operable individually or as a collective, there are no hard-coded connections that require dependencies from either direction.

A linear algebra approach was utilized to create the effort's algorithms using the Rust-based NAlgebra-GLM package. These Dead Reckoning algorithms are essentially dynamic models. The packet traffic over the network was analyzed using Wireshark to ensure proper functionality by analyzing the pieces of each PDU sent. The next step is to add more packet types to the system, increasing the diversity of effects that can be simulated. This action also takes another step towards fleshing the system out into a fully operable DIS system that could be used in live applications.

Future work on this effort would be to integrate the backbone of this system into a higher fidelity simulation network, acting as another machine within the simulated environment. By integrating a fully implemented ECS system into a preexisting exercise, it would display that this effort has led to a fully functional system using ECS and Rust.

6. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] "Using Serde to Serialize and Deserialize DIS PDUs," 2020.
[2] "A Data-Oriented ECS-based Design to Send DIS Packets," 2021.
[3] "Prediction and Convergence Calculations using Rust-based NAlgebra GLM," 2020.
[4] "IEEE Standard for Distributed Interactive Simulation – Application Protocols," *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*, pp. 1–747, 2012.
[5] "Serde," 2016, accessed 03-Nov-2020. [Online]. Available: https://serde.rs/
[6] "Entity-Component-System (ECS)," 2020, accessed 02-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Entity_component_system
[7] "Rust HECS crate site," 2021, accessed 17-June-2021. [Online]. Available: https://docs.rs/hecs/0.5.2/hecs/
[8] "Wireshark," 2021, accessed 6-Sep-2021. [Online]. Available: https://www.wireshark.org/
[9] "Distributed Interactive Simulation," 2020, accessed 15-Oct-2020. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_Interactive_Simulation
[10] "nalgebra," 2021, accessed 30-July-2021. [Online]. Available: https://nalgebra.org/
[11] "nalgebra-glm," 2021, accessed 31-July-2021. [Online]. Available: https://docs.rs/nalgebra-glm/0.15.0/nalgebra-glm/
[12] G. Palmer, *Physics for Game Programmers*. USA: Apress, 2005.

Author Biographies

NOAH W. SCOTT is a 1st Lieutenant stationed at Wright Patterson Air Force base. He is master's student at the Air Force Institute of Technology studying software engineering concepts, primarily with regards to Simulations and Game networking. His previous work was programming software which interfaced with DIS simulation technology within AFRL.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MAJOR RICHARD DILL is an Assistant Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a Ph.D. in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence. He can be contacted via email at richard.dill@afit.edu.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; Ph.D. 1999, Texas AM University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

VI. Conclusions

Through researching this novel method of implementing DIS simulation technology, we have explored how Data-Oriented Design can be used to design and operate a node that complies with IEEE DIS requirements. This new approach differs from the Object-Oriented methods by using the ECS style of organizing data into loosely connected lists through which the Entities are formed. All systems created for the software operates directly on lists of data, which eliminates the need to access specific object types and allows for additional systems to be added which query the preexisting lists.

Designing simulation nodes using a data-oriented approach in Rust enforces safe memory practice as well. The Rust compiler will not allow code that causes pointers to unallocated data, nor will it permit possible data races. This makes systems and programs less exploitable and less prone to error, an important trait for distributed simulations especially with regards to military application. By showcasing and further pursuing this research, simulation capabilities within the Air Force and the DoD will continue to increase options for creating simulated training environments.

6.1 Future Work

- Add additional functions to handle each packet type (i.e. Electronic Warfare signals, Fire/Detonation)
- Implement different types of nodes in a similar fashion, such as Surface Warfare

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
19–03–2022	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

AN ENTITY-COMPONENT SYSTEM BASED,
IEEE DIS INTEROPERABILITY INTERFACE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Noah W. Scott

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-22-M-060

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RQQD
Building 145
WPAFB OH 45433-7765
DSN 798-6556, COMM 937-904-6556
Email: James.Zeh@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RQQD

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In practice, there are several different methods of organizing data within a given software to fulfil its function. The method known as the Entity-Component System (ECS) is a software architecture where data components define entities. These components are stored as organized lists which are operated upon by systems to inject the system's desired behavior. Data is sent across the networks to communicate between simulation nodes as Protocol Data Units (PDUs). When sending PDUs across a network protocol, each simulation represents a common understanding of the world at the desired level of detail. DIS-compliant simulations are commonly written using an Object-Oriented Programming paradigm, where data is contained within objects and state is altered through exchanged method calls. The prototype is written using the Rust programming language. It depends upon nalgebra-glm, Serde, and Handy ECS crates for full functionality using matrix calculations, serialization/deserialization techniques, and simplified ECS capabilities respectively.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Douglas D. Hodson, AFIT/ENG
U	U	U	UU	44	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 4719

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18