Open-DIS: An Open Source Implementation of the DIS Protocol for C++ and Java

Don McGregor
Don Brutzman
Naval Postgraduate School, MOVES Institute.
700 Dyer Road, Bldg 246, Rm 265
Monterey, CA 93943
831-656-2149
{mcgredo | brutzman }@nps.edu
John Grant
Systems Technology, Inc
jgrant@systemstech.com

Keywords:
Distributed Interactive Simulation, DIS, Open-DIS, Open Source, Efficient XML Interchange, EXI

ABSTRACT: *The Distributed Interactive Simulation (DIS) protocol has long been used in military simulations, but no widespread open source C++ implementation has been made available to date. We have written an open source implementation in C++ and Java that we believe will result in a less duplicated effort in creating DIS simulations and that can serve as a platform for investigating modifications and extensions to the protocol. The Open-DIS Java and C++ implementations were created from an XML template language, and other language implementations can be created as needed. The C++ and Java implementations are capable of representing the data contained in protocol data units in DIS format, and the Java implementation can in addition represent the data in XML and Java object serialization formats. A schema for the XML format is also provided. Preliminary results show that the size of the information contained in DIS packets represented in XML and encoded with the draft version of the World Wide Web Consortium's Efficient XML Interchange format are comparable to that of the native binary DIS format. The project is royalty-free, open source and has a non-viral Berkeley Software Distribution (BSD) license. We describe the approach taken in creating the implementation, the outlines of the implementation, and the community established for its maintenance and improvement.*

1. Distributed Interactive Simulation

The Distributed Interactive Simulation protocol (DIS, IEEE-1278.x) [1] is a standard for binary exchange of information in military simulations. The interoperability of the standard relies primarily on a consistent format for information on the wire and agreed-upon enumerations of constant values. Anyone can obtain the IEEE-1278.x standard and implement a compliant DIS library using any programming language API they choose, so long as the API reads and writes data in the IEEE-1278.x format.

The Modeling, Virtual Environments, and Simulation (MOVES) Institute at the Naval Postgraduate School (NPS) has in the past implemented portions of the DIS protocol in Java, using a variety of techniques. DIS-XML [2] used an XML schema and Sun's Java for XML Binding (JAXB) 1.x [3] to generate code that could marshal and unmarshal data to XML, and added hand-written code to marshal and unmarshal from the IEEE-1278.x binary format. This worked well, and the ability to marshal to XML format was found to be particularly useful. We have used Extensible Messaging and Presence Protocol (XMPP) chat servers with DIS Protocol Data Units (PDUs) in XML format to bridge wide area networks across firewalls over chat [2], and have used XML as an archiving format for DIS traffic. We have made use of DIS-XML in other projects, such as AUV Workbench [4] and Xj3D [5] with good results, and have made use of MATLAB's Java loading facility to use the Java implementation within that tool.

The drawbacks to the DIS-XML approach primarily affected the protocol developers. These included the necessity of first writing the XML schema, and then having to write the code to marshal and unmarshal the information to IEEE-1278.x format. Certain coding idioms used in the DIS PDUs were also not very easily expressed in XML schema. While JAXB is a useful tool for creating Java code from XML schemas, similar tools for C++ were found wanting, either because they required a license or because the code they generated was thought to be impenetrable to typical C++ programmers. The API generated in the C++ code was not necessarily similar to the API of the Java code, and it was thought that a similar API for both Java and C++ would shorten programmer learning curves.

An open source C++ implementation was desired for the MOVES Institute's Delta3D game engine [6], so the problems encountered in generating C++ code from the existing XML schema were disappointing.

Most of the C++ DIS implementations to date have either been commercial products or internal to a specific corporate project. Both of these approaches have drawbacks. Commercial products often have good performance and support, but some modeling and simulation projects are unwilling to encumber their efforts with a dependency on a commercial license, in part because of the very long software lifecycles in Department of Defense (DoD) projects. A project that lasts for twenty years may outlast some of the companies providing the software. This may leave a project in limbo, or at least require a rewrite to accommodate the API of a different DIS implementation. On the other hand projects that use internal DIS implementations very often wind up duplicating previous work rather than using developers to add value with innovative and new features. We want those implementing DIS projects to stop working on the DIS portion and start working on the value-added portions of the project.

2. Open-DIS

Open-DIS is an effort to make available to the modeling and simulation community a full implementation of the DIS protocol in both C++ and Java. It is open source and carries a Berkeley Software Distribution (BSD) license. The BSD license is notable for its non-viral nature; there is no requirement for any modifications written by users to be released, and code that uses the library does not become open source. Users can simply make use of and modify the code as they see fit, while retaining all rights to the code they write. If the user improves the Open-DIS implementation they are encouraged to send the changes back to be incorporated in the main project, but this is not a legal requirement.
Open-DIS is a cooperative effort to create a complete and correct DIS implementation. It consists of

- An abstract description of the data in DIS Protocol Data Units (PDUs), expressed in XML
- C++ and Java language implementations of the PDUs
- Code to marshal and unmarshal the Open-DIS objects to various formats
- An XML schema for the DIS protocol
- A community to support and extend the implementation.

2.1 Abstract description of PDUs using XML

The PDUs in the DIS specification consist of a few dozen discrete binary data packets formatted in accordance with the DIS specification. Each PDU contains fields for information such as an entity's velocity and position. The exact location and format of the fields within the packet is defined by the DIS standard.

In C++, Java, and most other object-oriented languages, a PDU can be modeled as a class that has:

- Instance variables for each field in a PDU
- Methods to get and set each field value
- Constructor(s)
- Methods to transform the data in the object to the DIS format, and to set instance variable fields from the values contained in DIS PDUs that are received from the network

If we begin with an abstract description of the fields in a DIS PDU—including their type and position in the PDU--that description can be processed to generate whatever computer language code is desired, and multiple computer language implementations can be generated from a single abstract description. For example, we may have an abstract description of an entity velocity field in a PDU that specifies the velocity consists of three floating point values in the order (x, y, z). We can use this information to generate a source code implementation in either Java or C++ that include the object-oriented features described above. A Java or C++ implementation of the DIS PDUs is over 20,000 lines of code, so creating a template from which we generate programming language code can save a significant amount of programming time, and in the end is likely to be more bug-free.

We chose to use a self-defined XML dialect to create the abstract description of PDUs rather than the XML schema used in DIS-XML. The XML dialect we used is simpler than XML schema. It also contains more information about the DIS protocol than XML schema, and this allows the creation of code that more closely resembled what would be used in a hand-written implementation. The protocol description language ASN.1 also could have been used, but we preferred to retain complete control of the source code generated; an XML dialect seemed a more direct path toward that goal. This approach also allowed the marshalling of objects to XML or Java object format.

The DIS standard describes four types of fields in PDUs:

• Primitive values: integers of various lengths, signed or unsigned, and single or double precision floating point.
• Another, smaller subunit of the PDU that is treated as a reusable object
• An array of fixed length
• A list of variable length

The objects contained within PDUs include fields such as the velocity, which consists of three float values that can be treated as a single object. The objects are defined in section 5.2 of IEEE-1278.1, “Basic data types and records.”

Arrays of fixed length are used for character and primitive type data arrays.

Variable length lists contain zero or more repetitions of an element, very often an object type; they are typically preceded (not necessarily immediately) by a field that holds a count of how many items are in the list. Variable length lists are one area where a custom XML dialect for describing DIS is superior to XML schema. XML schema does not have a way for meta-information to be easily added to the schema to describe the relationship in the serialized format between the count field and the variable length list field, while adding this information is straightforward in a custom XML dialect. The meta-information can be used to generate better a language implementation.

The DIS specification describes the PDUs in a way that is amenable to description using an object-oriented inheritance hierarchy. Section 5.3.2 of IEEE-1278.1 describes PDUs and their grouping into “PDU families.” For example, the Entity State PDU (ESPDU) and Collision PDU are both members of the Entity Information/Interaction PDU family, and can be modeled as a class/subclass relationship. The ESPDU and Collision PDUs share several fields; this can be modeled using standard object-oriented techniques by using a common abstract superclass of PDU, an intermediate level abstract class of EntityInformationFamilyPdu that inherits from Pdu, and two concrete subclasses classes, EntityStatePdu and CollisionPdu, that inherit from EntityInformationPdu.

Figure 1 shows a simple example of the XML description file, in this case an excerpt from the PDU header that is common to all PDUs.

```
<class name="Pdu"
inheritsFrom="root" comment="The
superclass for all PDUs.">

<attribute
name="protocolVersion"
comment="The version of the
protocol. 5=DIS-1995,6=DIS-
1998.">
<primitive type= "unsigned byte"
defaultValue="6"/>
</attribute>

<attribute name="exerciseID"
comment="Exercise ID">
<primitive type= "unsigned byte"
defaultValue="0"/>
</attribute>

<attribute name="pduType"
comment="Type of pdu, unique for
each PDU class">
<primitive type="unsigned
byte"/>
</attribute>
…
```

Figure 1. Excerpt From PDU Description File

The XML dialect is straightforward. The comment attribute can be extracted to supply comments to the generated language source code, and the default value can be used to specify the value to which the fields should be set in the constructor. The "inheritsFrom" attribute of the class element can be used to define the class inheritance hierarchy described earlier.

The XML document that represents the DIS standard must be written by a programmer who has read and interpreted the IEEE standard. This process is somewhat less complex than writing an XML schema, and the document contains more information relevant to code generation.

2.2 Java and C++ Language Implementations. A small program was used to process the XML description and generate Java and C++ source code. Enough information is present in the XML file to create an implementation that is nearly as complete as hand-written code. The generated code does not, however, completely and correctly describe every PDU. The abstract description was unable to correctly describe about fifteen of the 65 or so PDUs. The work remaining to be done on those PDUs ranges from minor to significant. The defects in the generated code do not necessarily have to be fixed in the generator. The generated code is an artifact in its own right, and can be checked into source code control for manual modification, rather than attempting to get the code generator exactly right.

As mentioned earlier, the PDUs generated by Open-DIS are arranged in a class inheritance hierarchy that is the same for C++ and Java. A portion of the class inheritance diagram that includes the Entity Information/Interaction PDU family is shown in figure 2.

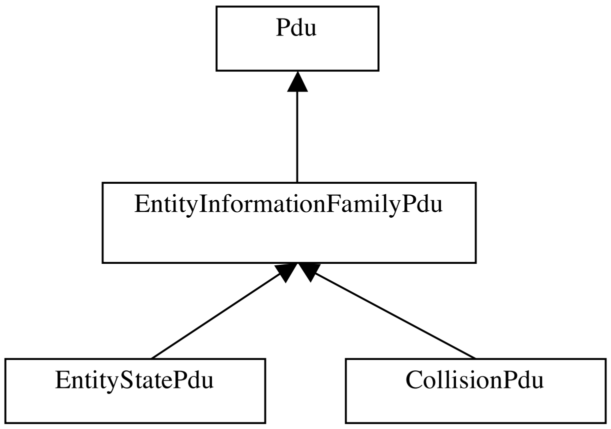


Figure 2. Partial DIS Class Inheritance Hierarchy for Entity Information/Interaction Family

The other PDU families are arranged in a similar inheritance hierarchy. All PDUs eventually inherit from the Pdu class.

A list of the implemented PDUs organized by PDU family is shown in figure 3. The PDUs that are not complete or not yet fully implemented are shown in bold.

`Pdu:`
`EntityInformationFamilyPdu: EntityStatePdu, FastEntityStatePdu, CollisionElasticPdu, CollisionPdu, EntityStateUpdatePdu`

`WarfareFamilyPdu: FirePdu, DetonationPdu`
`LogisticsFamilyPdu: ServiceRequestPdu, ResupplyOfferPdu, ResupplyReceivedPdu, ResupplyCancelPdu, RepairCompletePdu, RepairResponsePdu`

`SimulationManagementFamilyPdu: StartResumePdu, StopFreezePdu, AcknowledgePdu, ActionRequestPdu, ActionResponsePdu, DataQueryPdu, SetDataPdu, DataPdu, EventReportPdu, CommentPdu, CreateEntityPdu, RemoveEntityPdu`

`DistributedEmissionsFamilyPdu: ElectronicEmissionsPdu, DesignatorPdu,` **`UaPdu`**`, IffAtcNavAidsLayer1Pdu, IffAtcNavAidsLayer2Pdu, SeesPdu`

`RadioCommunicationsFamilyPdu:` **`TransmitterPdu`**`,` **`SignalPdu`**`, ReceiverPdu, IntercomSignalPdu, IntercomControlPdu`

`MinefieldFamilyPdu: MinefieldStatePdu,` **`MinefieldQueryPdu`**`,` **`MinefieldDataPdu`**`, MinefieldResponseNackPdu`

EntityManagementFamilyPdu: **AggregateStatePdu, IsGroupOfPdu, TransferControlRequestPdu,** IsPartOfPdu

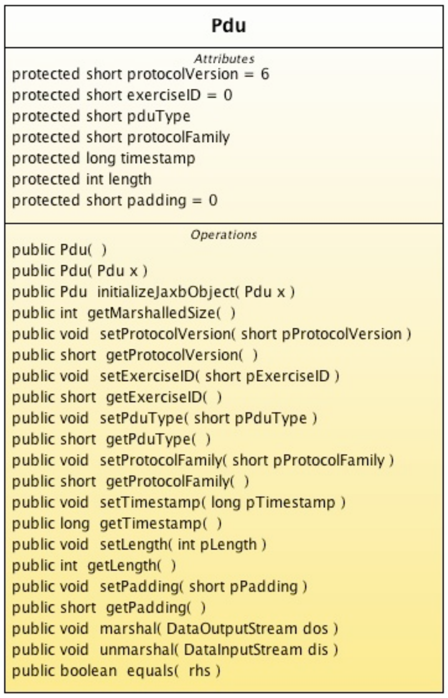
SyntheticEnvironmentFamilyPdu: **EnvironmentalProcessPdu, GriddedDataPdu,** PointObjectStatePdu, LinearObjectStatePdu, ArealObjectStatePdu

SimulationManagementWithReliabilityFamilyPdu: StartResumeReliablePdu, StopFreezeReliablePdu, AcknowledgePduReliablePdu, ActionRequestReliablePdu, ActionResponseReliablePdu, DataQueryReliablePdu, **SetDataReliablePdu,** DataReliablePdu, **EventReportReliablePdu, CommentReliablePdu,** CreateEntityReliablePdu, RemoveEntityReliablePdu, RecordQueryReliablePdu, **SetRecordReliablePdu,** RecordReliablePdu

Figure 3. Open-DIS PDUs

Both the Java and C++ code are generated with class definitions, instance variables for each of the fields, methods to get and set the field values, and methods to marshal the objects to IEEE-1278.x format. The Java objects can also marshal themselves to XML or Java object serialization format. Supporting objects for fields contained within the PDUs are also generated, such as the velocity vector objects. These are used as return values and parameters in get() and set() methods.

A Unified Modeling Language (UML) diagram for the top-level Java PDU class is shown in figure 4.



```
Pdu

Attributes
protected short protocolVersion = 6
protected short exerciseID = 0
protected short pduType
protected short protocolFamily
protected long timestamp
protected int length
protected short padding = 0

Operations
public Pdu( )
public Pdu( Pdu x )
public Pdu  initializeJaxbObject( Pdu x )
public int  getMarshalledSize( )
public void  setProtocolVersion( short pProtocolVersion )
public short  getProtocolVersion( )
public void  setExerciseID( short pExerciseID )
public short  getExerciseID( )
public void  setPduType( short pPduType )
public short  getPduType( )
public void  setProtocolFamily( short pProtocolFamily )
public short  getProtocolFamily( )
public void  setTimestamp( long pTimestamp )
public long  getTimestamp( )
public void  setLength( int pLength )
public int  getLength( )
public void  setPadding( short pPadding )
public short  getPadding( )
public void  marshal( DataOutputStream dos )
public void  unmarshal( DataInputStream dis )
public boolean  equals(  rhs )
```

Figure 4. UML Diagram for Pdu Class

The instance variables correspond to the defined fields in the IEEE-1278.x specification for the PDU header. All PDUs inherit from this class. The marshal() and unmarshal() methods are used to convert the Java or C++ object to and from the binary format specified by the IEEE 1278.x standard.

The C++ code is very similar to the Java code. The C++ implementation includes separate header (.h) and implementation (.cpp) files, a destructor method to reclaim dynamically allocated memory, and an additional “const” get method, which returns a non-modifiable reference to PDU fields.

A UML class diagram for the more complex Fire PDU is shown in figure 5. This PDU contains data relating to the firing of a munition.

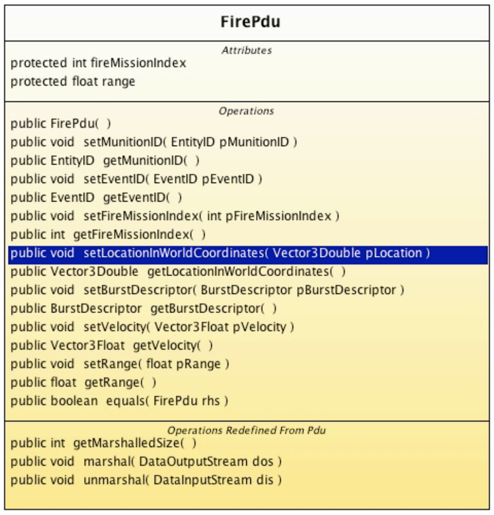

FirePdu
Attributes
protected int fireMissionIndex
protected float range
Operations
public FirePdu( )
public void setMunitionID(EntityID pMunitionID)
public EntityID getMunitionID( )
public void setEventID(EventID pEventID)
public EventID getEventID( )
public void setFireMissionIndex(int pFireMissionIndex)
public int getFireMissionIndex( )
public void setLocationInWorldCoordinates(Vector3Double pLocation)
public Vector3Double getLocationInWorldCoordinates( )
public void setBurstDescriptor(BurstDescriptor pBurstDescriptor)
public BurstDescriptor getBurstDescriptor( )
public void setVelocity(Vector3Float pVelocity)
public Vector3Float getVelocity( )
public void setRange(float pRange)
public float getRange( )
public boolean equals(FirePdu rhs)
Operations Redefined From Pdu
public int getMarshalledSize( )
public void marshal(DataOutputStream dos)
public void unmarshal(DataInputStream dis)

Figure 5. UML diagram for Fire PDU

The FirePdu class demonstrates the use of objects in get() and set() methods. The getLocationInWorldCoordinates() method returns a Vector3Double, an object containing three double precision floating point values. The FirePdu class overrides some methods defined in classes higher in the class hierarchy, including the marshal and unmarshal methods. The class also allows access to all the public methods of classes it inherits from, including the Pdu class discussed earlier.

Java uses garbage collection (GC) to reclaim memory used in objects that are no longer referenced, in contrast to the manual memory management typically used in C++ programs. However, GC can be problematic in real time applications because it can impose an asynchronous and significant CPU load on the host that is running the program. In this situation it makes sense to minimize the number of Java objects generated to reduce the GC load.

The Java PDUs contain fields that are themselves objects, such as the position field in the FirePdu above. While desirable from a software engineering and programmer convenience standpoint, this also increases the number of Java objects that are generated, which places a greater load on the Java GC subsystem when the time comes for memory to be reclaimed.

Research has shown that the vast majority of PDUs in a typical DIS exercise are Entity State PDUs. [7] If the Entity State PDUs are made more memory-efficient, most of the garbage generated by the high-volume ESPDUs will be eliminated while the programmer will retain a convenient API for the other PDUs. To this end we have also provided a stripped-down "Fast ESPDU", which contains only the bare minimum possible of fields represented as objects. All fields in the PDU are instead represented as primitive types that do not require allocation on the Java heap, and are therefore not processed by the GC subsystem. In high performance real time implementations it may be worthwhile to use this alternative implementation.

2.3. Marshalling and Unmarshalling

The Open-DIS project also contains supporting code that allows programmers to marshal and unmarshal PDUs to several formats of their choosing. The C++ implementation marshals to the IEEE-1278.x format. The Java implementation marshals to IEEE-1278.x, XML, and Java object serialization formats.

Almost any DIS application will need to read IEEE-1278.x information from the network and convert it into an accessible format for internal application use. The Java code provides the PduFactory class to unmarshal PDUs. PduFactory implements the "Gang of Four" [8] factory object pattern. A UML diagram of PduFactory is shown in Figure 6.

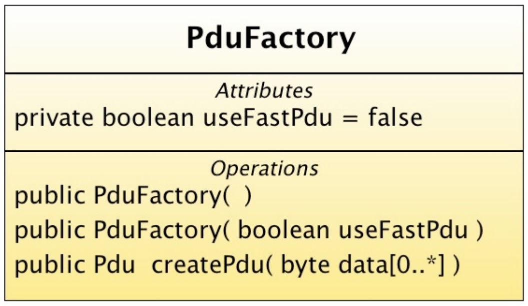

PduFactory
Attributes
private boolean useFastPdu = false
Operations
public PduFactory( )
public PduFactory(boolean useFastPdu)
public Pdu createPdu(byte data[0..*])

Figure 6. UML Diagram for PduFactory

The createPdu() method accepts an array of bytes in IEEE-1278.x format and returns the concrete PDU subclass that corresponds to that byte array. A network socket typically reads an array of bytes from the network; we pass the bytes to the method createPdu() in an instance of the PduFactory class and receive back an instantiated Pdu object. The Pdu instance will be a concrete subclass such as EntityStatePdu, CollisionPdu, or FirePdu. The exact class returned can be determined at runtime in Java by the instanceof operator, or by calling the getPduType() method to return the appropriate enumerated value in either Java or C++. If GC performance is a concern, create a factory object using the alternate constructor. This factory instance will return FastEspdus that require minimal object generation.

The C++ code uses a similar factory pattern to create new PDU objects from byte arrays.

The Open-DIS Java implementation is capable of marshalling PDU objects to IEEE-1278.x format, XML, or Java object serialization format. Figure 7 illustrates code that marshals Java language PDUs to IEEE-1278.x format.

```
EntityStatePdu espdu =
    new EntityStatePdu();
ByteArrayOutputStream baos =
   new ByteArrayOutputStream();
DataOutputStream dos =
    new DataOutputStream(baos);
espdu.marshal(dos);
byte[] ieeeData =
   baos.toByteArray();
```

Figure 7. Marshalling Java PDUs to IEEE-1278.x Format

A typical use for this code would be to create a new PDU, set field values such as location, orientation and velocity, and then place it into IEEE-1278.x format and send it to other hosts over the network.

Each PDU object and all the objects contained within them also implement the Java Serializable interface, which allows them to be marshaled to the standard Java ObjectStream class. An example of this is shown in figure 8.

```
EntityStatePdu espdu =
  new EntityStatePdu();
FileOutputStream fos =
  new FileOutputStream("pdus");
ObjectOutputStream   oos   =   new
ObjectOutputStream(fos);
oos.writeObject(espdu);
```

Figure 8. Marshalling Open-DIS PDUs to Java Object Serialization Format

This marshals the information contained in the entity state PDU to a binary archive format supported by Sun Microsystems. The ability to send PDU objects in this format can be useful when communicating with other Java programs or Java-specific tools.

The ability to marshal Open-DIS objects to XML was more difficult to implement. There is no agreed-upon standard for an XML representation of DIS. Creating an XML schema directly is a difficult and error-prone process. Instead of writing a schema from scratch or reusing the partial schema we developed in DIS-XML, we used the JAXB 2.x release, which is capable of processing either JavaBeans-compliant code or source code marked up with "annotations" to generate an XML document and schema.

JavaBeans-compliant code conforms to a few simple coding standards, such as having methods that follow the idiom getXXX() and setXXX() for every instance variable named XXX. Java 1.5 also introduced annotations to allow markup of code with metadata that can be used by external processors [9]. JAXB 2.x (included in the Java 1.6 release) has a number of annotations that can be used to describe the mapping between Java code and an XML document. For example, the code excerpt in Figure 9 below shows a field value specified as being described by an XML attribute when marshaled to an XML document, rather than an XML element.

```
@XmlAttribute
public int getCapabilities()
{ return capabilities;
}
```

Figure 9. Annotations for use with XML

Objects marked up by annotations can be directly marshaled to XML in less than ten lines of code. An example of a marshaled Fire PDU is shown in figure 10.

```
<pdus
xmlns:xsi=http://www.w3.org/2001/XMLSchem
a-instance xsi:type="firePdu" range="0.0"
fireMissionIndex="0" timestamp="0"
protocolVersion="6" protocolFamily="2"
pduType="2" padding="0" length="0"
exerciseID="0">
<firingEntityID site="0" entity="0"
application="0"/>
<targetEntityID site="0" entity="0"
application="0"/>
<burstDescriptor warhead="0" rate="0"
quantity="0" fuse="0">
<munition subcategory="0" specific="0"
extra="0" entityKind="0" domain="0"
country="0" category="0"/>
</burstDescriptor>
```

```
<eventID site="0" eventNumber="0"
application="0"/>
<locationInWorldCoordinates z="0.0"
y="0.0" x="0.0"/>
<munitionID site="0" entity="0"
application="0"/>
<velocity z="0.0" y="0.0" x="0.0"/>
</pdus>
```

Figure 10: Fire PDU in XML Format

A schema that matches the XML representation can also be easily generated. We have included one in the Open-DIS package available for download.

2.5 Networking Support. The Java implementation includes example network source code to send and receive DIS PDUs in IEEE format on a multicast socket in the edu.nps.moves.examples package. The EspduSender and EspduReceiver classes illustrate one way to send and receive entity state PDUs.

The C++ implementation relies on the open source HawkNL network library and also contains example code for sending and receiving PDUs. Any networking package could be used in place of HawkNL, including WinSock or Berkeley sockets. The difficult part is getting the information into IEEE-1278.x format; once that is done the networking code is not extensive.

2.6 Community The DIS implementation is not complete, and it would be helpful to have the assistance of others in completing the library and adding new features. Doubtless bugs exist in the current Java and C++ implementations. Open source projects are more than a one-time code dump; they need a supporting infrastructure to maintain and improve the code, which is ultimately a social process.

To this end we have set up a site at sourceforge.net, the leading repository of open source projects. Sourceforge.net maintains a permanent site for source code control, forums, mailing lists, code releases, mirrors, and the full spectrum of development tasks. The URL for the Open-DIS project is https://sourceforge.net/projects/open-dis.

3. Performance

Previous experience has shown that the performance of protocols that use User Datagram Protocol (UDP) as transport is dominated by the number of packets per second that arrive on the network interface. Typically a TCP/IP stack generates an operating system interrupt for each UDP packet that arrives. Some time is required to handle each interrupt. This can easily lead to a situation in which the network interface device driver, TCP/IP stack, and operating system are busy responding to interrupts rather than parsing code in user space. Benchmarks that count the number of DIS UDP packets per second processed by a host are somewhat suspect; they often measure the tuning and efficiency of the operating system, TCP/IP stack, network socket options, and the characteristics of the sending process rather than any inherent characteristics of the DIS protocol implementation.

Nonetheless, we benchmarked a 3 Ghz Pentium 4 CPU running CentoOS 5 Linux processing about 5000 DIS UDP/multicast packets per second using the C++ implementation. Running the Java implementation on a 2 Ghz, Intel Core Duo with Apple OS X v. 10.5, about 1,000 entity state PDUs per second were received and processed.

We have also implemented a PDU logging and playback utility. The utility was able to log approximately 1,000 PDUs per second to XML on a 2 GHz Intel Core Duo running OS X v 10.5 on a MacBook Pro laptop.

The World Wide Web consortium (W3C) has chartered a working group to create a standard for Efficient XML Interchange (EXI) format [10]. EXI relaxes the requirement for XML to be in text format in exchange for a more compact and faster to parse representation. We have encoded some of the XML documents containing an entity state PDU created by Open-DIS in accordance with the draft EXI specification. Some preliminary results are shown in figure 11.

IEEE Binary	Gzipped IEEE	XML	Gzipped XML	EXI XML
144	127	1387	548	145

Figure 11. Comparative Entity State PDU file sizes, in bytes

Interestingly, the EXI-encoded documents wound up being essentially the same size as the original IEEE 1278.x format. The exact size of EXI documents varies somewhat depending on the data, but the results so far show that EXI documents are consistently the same size or smaller than the original binary IEEE 1278.x format. The EXI format retains in compact form the XML markup inherent to any XML document, and can be parsed by any standards-compliant EXI processor and converted back to the original XML.

The ability to represent data protocols in an XML-friendly format that is essentially the same size as the original binary format may reduce the "data silo" problem of binary data formats. Protocols could send data in EXI format for little or no cost in bandwidth, while retaining full access to the XML tools environment and the data interoperability inherent in XML. The cost for this is somewhat more processing to put the data into EXI format and decode it on the receiving side.

4. Conclusions and Future Work

The Open-DIS library promises to simplify the implementation of new DIS-related software by removing licensing obstacles and barriers to entry. Use of the library should match well with the very long project lifecycles of DoD software, reduce duplicated effort, and focus programmers on adding new value rather than re-implementing old functionality yet again.

It is notable that we chose to use a self-defined XML dialect rather than XML schema. This choice was made in part because the XML schema could not easily represent the relationships between some DIS protocol fields. A system for annotating XML schemas with additional information might have been helpful in overcoming this problem.

The process of reading the DIS standard and creating the abstract description of the protocol is tedious and error-prone. It would be helpful for implementers if the protocol were defined in an unambiguous and computer-friendly format to begin with. This would allow computer language implementations to be created directly from the standard, rather than by having a human read and interpret the standard. This would lower the barriers to entry in creating implementations of protocol standards and result in more error-free protocol implementations. Considering the compactness of EXI data representations, one possibility is to define PDUs in terms of XML documents. This would allow direct representation of data in XML format, which greatly expands the interoperability of the standard.

Further work can be done in completing the PDU implementations, and in creating supporting software such as dead reckoning software, client-side finite state machines, enumerated types, and more. DIS enumerations are a particularly promising area for further work, since the DIS enumerations database is in the process of being converted to XML. This should facilitate the automated generation of enumeration data structures.

The Open-DIS code should provide a stable foundation for this work. An obvious extension would be to integrate Open-DIS with an HLA RTI to create an HLA-DIS gateway, perhaps using the Portico open source RTI [11]. Open-DIS may facilitate integration with other networking standards, such as massively multiplayer online game engines.

1. IEEE Standard 1278.1-1995 (and revisions), Standards Committee on Interactive Simulation (SCIS) of the IEEE Computer Society, Approved September 21, 1995.
2. McGregor, Don, Brutzman, Don, Blais, Curtis, Armold, Adrian, Falash, Mark, Pollak, Eytan: DIS-XML: Moving DIS to Open Data Exchange Standards, Spring Simulation Interoperablity Workshop, 2006.
3. Sun Microsystems, https://jaxb.dev.java.net, downloaded Feburary 1 2008.
4. Jeffrey Weekley, Don Brutzman, Anthony Healey, Duane Davis and Daryl Lee, "AUV Workbench: Integrated 3D for Interoperable Mission Rehearsal, Reality, and Replay" *2004 Mine Countermeasures & Demining Conference: Asia-Pacific Issues & Mine Countermeasures (MCM) in Wet Environments*, Australian Defence Force Academy, Canberra Australia, 9-11 February 2004.
5. Web3D Consortium, http://www.xj3d.org, downloaded Febuary 1, 2008.
6. Delta3D Open Source Gaming & Simulation Engine, http://www.delta3d.org, downloaded July 2, 2008

7. Macedonia, Michael, Zyda, Michael, Pratt, David, Brutzman, Donald, Barham, Paul: *Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments*, IEEE Computer Graphics, September 1995, Vol. 15, No. 5, pp. 38-45.
8. Gamma, Eric, Helm, Richard, Johnson, Ralph, Vlissides, John: Design Patterns: Elmements of Reusable Object-Oriented Software. 1995 Addison-Wesley
9. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, retrieved 6/15/2008
10. http://www.w3.org/XML/EXI, retrieved 6/15/2008
11. http://porticoproject.org, retrieved 2/2/20008