ARL-TR-8274 ● JAN 2018

US Army Research Laboratory

US Army Research Laboratory Visualization Framework Architecture Document

by Chien Hsieh and Andrew Toth

ARL-TR-8274 ● JAN 2018

US Army Research Laboratory

US Army Research Laboratory Visualization Framework Architecture Document

by Chien Hsieh and Andrew Toth
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE					*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* January 2018	**2. REPORT TYPE** Technical Report	**3. DATES COVERED (From - To)** 1 January–30 September 2017
4. TITLE AND SUBTITLE US Army Research Laboratory Visualization Framework Architecture Document		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Chien Hsieh and Andrew Toth		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory Computational and Information Sciences Directorate (ATTN: RDRL-CIN-T) 2800 Powder Mill Road Adelphi, MD 20783-1138		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-8274
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES
Approved for public release; distribution is unlimited.

14. ABSTRACT
Visualization of network science experimentation results is generally achieved using stovepipe solutions tailored to specific experiments and performance metrics. Based on ZeroMQ, the US Army Research Laboratory (ARL) Visualization Framework presents a language-agnostic platform-independent approach to connecting data published by data sources to visualizations using a publish/subscribe mechanism and ZeroMQ. The framework provides for automated discovery of data sources by the visualization without prior knowledge of the data sources. This report documents the ARL Visualization Framework system design and specific details of its implementation.

15. SUBJECT TERMS
visualization, experimentation, ZeroMQ, publish/subscribe, network science, NSRL, service discovery

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Chien Hsieh
Unclassified	Unclassified	Unclassified	UU	26	**19b. TELEPHONE NUMBER (Include area code)** (301) 394-2365

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

Contents

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

Scientific visualizations are often developed to support a specific experiment or series of experiments. These efforts, which can often balloon into full-blown development projects, are often discarded as the research diminishes or when the technology du jour reaches a tipping point. Reuse of an existing visualization often requires researchers to write adapters to existing software codes or rewrite large sections of code. This report describes a US Army Research Laboratory (ARL) project to develop a generalized approach to developing data sources and visualizations such that they may be combined in new ways and used by new technologies as they are developed.

The ARL Visualization Framework (ARLVF) provides an infrastructure for a variety of visualization and data-producing tools to collaborate in order to create a platform to monitor experiments and visualize results for researchers. A principle goal of the ARLVF is to provide a general and flexible approach to creating applications and integrating tools into this visualization platform.

Since the ARLVF was initially designed and implemented in the fall of 2015 (version 0.1),[1] changes have been introduced as part of the deployment of several data visualization applications in the Network Science Research Laboratory.[2] Those changes were necessary to adapt to the new requirements that emerged from the initial use of the visualization applications. This report details the architecture and design of the current version of ARLVF (version 0.2).

2. Analysis

The following sections discuss the important requirements and use cases that helped drive the changes in the ARLVF.

2.1 Requirements

The general requirements from version 0.1 of ARLVF continue to apply. Those requirements, which specified ease of use, language agnostic, operation system agnostic, and flexible deployment environment, are still very relevant. The following main requirements for version 0.2 address specific operational needs:

- Query options. Visualization applications should have a mechanism through which to describe what and how the requested data are to be procured. For example, it may want to receive only data that satisfies certain filter criteria, or data are to be in JavaScript Object Notation

(JSON) format and the like. There was no process in the previous version to accomplish this function.

- Synchronous request response. In version 0.1, visualization applications were needed to implement asynchronous callback functions to receive requested data. It would be helpful for ARLVF to allow synchronous responses in order to simplify the processing at the visualization applications.
- Alternative data packaging. The original implementation of ARLVF used the Google Protobuf library to serialize and deserialize data as they are passed between ARLVF applications. There have been instances where a different version of Protobuf was installed and used in the environment where ARLVF was operating, and the version conflict prevented the ARLVF from functioning correctly.

2.2 Use Cases

The following are possible use cases for using the ARLVF, in terms of the way data are made available from the data producer:

- On demand. A visualization application requests and receives data from a data producer on an as-needed basis. The produced data can be received in a single data transmission (nonstreaming) or multiple transmissions (streaming). Optional parameters can be embedded in request messages.
- Broadcast. A visualization application requests to receive data that are being broadcast by a data producer in an ongoing basis. These data are being produced and published regardless of whether there is any active consumer. Once the data subscription begins, the data will be streamed to the visualization application until a request to stop is issued by the visualization application.
- Directed. A data producer application streams data directed at a visualization application. This use case differs from the previous two in that the data are pushed from the data producer to the visualization application.

3. Architecture

The ARLVF is a platform that consists of a collection of libraries, executables, and utility modules (Fig. 1). The ARLVF as a whole serves as platform for the development and deployment of applications that easily provide and consume data and will ultimately lead to effective visual presentation of the data.

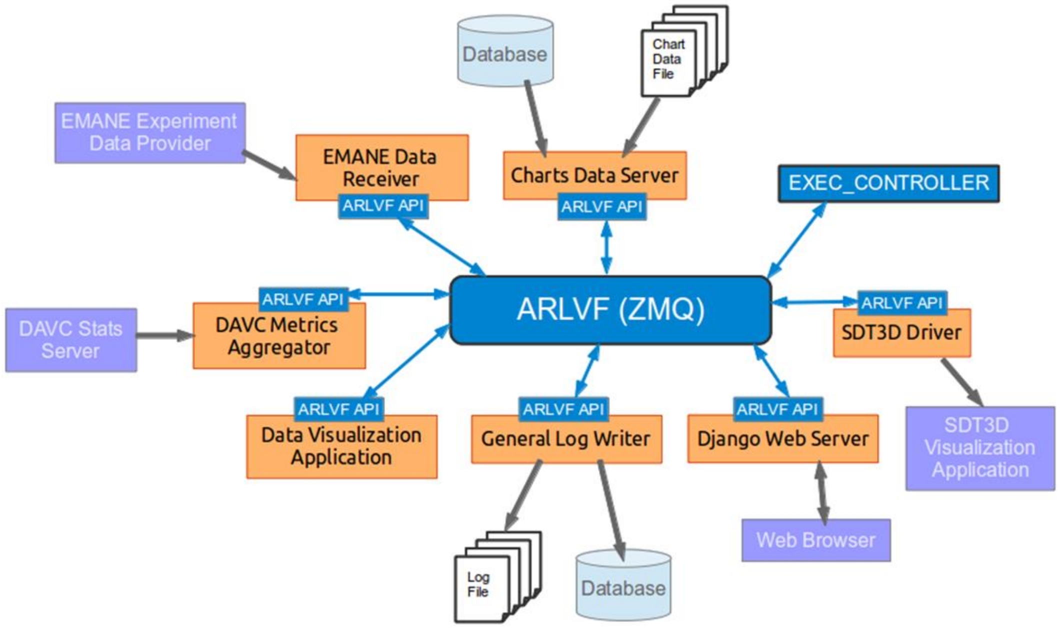

Fig. 1 ARLVF system architecture

Figure 1 illustrates the ARLVF in the context of a distributed visualization system of data sources and providers. The boxes in blue are the core infrastructure components provided by the ARLVF source code bundle. The boxes in orange are examples of applications that are developed by the user to work in the ARLVF environment. These applications can communicate with other resources or applications that can provide or visualize data. The following sections discuss the main components and concepts of ARLVF.

3.1 ZeroMQ

Consistent with version 0.1, ZeroMQ serves as the central messaging backbone of ARLVF. Specifically, it enables and maintains communication among ARLVF applications. The ARLVF utilizes the model of Pub-Sub socket pattern with a message broker.

3.2 Executive Controller

The Executive Controller is the application that monitors all applications participating in the framework. Furthermore, it serves as the data broker of the messages that traverse between the applications. It must be running and active for the framework and all of the ARLVF applications to operate properly. The purpose for making it a data broker is 2-fold. First, it serves as the central point of contact for all applications that produce and/or consume data in the system. Therefore, the applications are not required to connect directly to any other peer applications to send or receive data. Second, it provides a consolidated view of all the active ARLVF applications in any given time. In runtime the Executive Controller binds to 2 ZeroMQ sockets, one PUB and the other SUB.

3.3 VFAdapter

The VFAdapter is the main application programming interface (API) through which ARLVF applications (created by application developers) are able to communicate with each other. The VFAdapter is designed to be instantiated and contained in the ARLVF application. It provides the following framework functions for a given application module:

- It handles capabilities of the ARLVF application and sends and receives discovery messages.
- It provides interface functions for the ARLVF application to send messages. It will also invoke callback functions, to be defined by the contained ARLVF application, to handle processing of received messages.

3.4 Capability

Capability is a logical representation of a resource or facility that can provide or consume data. For example, a data provider capability can be a file of comma-separated values or a live stream of JSON data. Similarly, a data consumer capability can be an application that writes data to a log or display data to a bar chart. An ARLVF application may contain one or more capabilities, and the capabilities can be discovered by other ARLVF applications.

4. Message Design

4.1 General Structure

All messages sent by an ARLVF application module are “multipart” messages. Specifically, there are 3 parts: topic, metadata, and payload. ZeroMQ ensures that all or none of the parts are delivered to the recipient as one single transaction. Therefore, the multipart message structure is a convenient and effective way for ARLVF to partition its messages while guaranteeing the transactional integrity of each message.

The topic is the first part of the message and is a string with a hierarchical naming convention where each element in the name hierarchy is separated by a dot (“.”). The topic string specifies the characteristics about the content of the payload. The topic of a message allows ZeroMQ to apply filters when it determines whether to deliver the message based on the subscription filter. The following are 2 example topics:

- Discovery
- Request.96c239acc-cd52-52f9-229c-f4bae98bd628

The metadata are JSON data that describe the nature of the message. The following is an example of a metadata segment:

```
{
 "origin":"e9a2a3ea-bb28-11e6-96f7-d4bed98ac412",
 "msg_id":"09d298dc-ce50-25f9-47a2-e5cde07bd628",
 "correlation_id":"13a071ce-bb28-09c2-23b9-c6abd98cd920"
 "cap_uuid":"23b071bd-cc39-12d5-42c9-d7bd98ee819",
 "options":"user specified string",
}
```

Table 1 lists the metadata fields and provides a description along with the source of the field value.

Table 1 List of metadata fields and descriptions caption

Field	Description	Set/generated by
origin	UUID of the ARLVF application module. Same as module_uuid of its own discovery message. Randomly generated and set once during application startup.	Framework
msg_id	UUID of the message. Randomly generated.	Framework
correlation_id	Correlation ID. Value is to be populated only if this message is result of or related to an earlier message. In which case, correlation_id should be the msg_id of such earlier message. In other instances, it is blank.	Application
cap_uuid	Capability UUID. Populated when in a data message to be the cap_uuid of the data producer. Otherwise, it is blank.	Application
options	Optional string data field for application to specify parameters that accompany this data request. Default is blank.	Application

Notes: UUID = universally unique identifier; ID = identifier.

The payload is serialized binary data representing the payload of the message. The format and structure of payload is dependent on message type and applications.

4.2 Message Types

The following sections discuss the message types that ARLVF supports.

4.2.1 Discovery Message

On a regular interval, each ARLVF application sends a discovery message to the framework. This is essentially a keep-alive message to announce its continued presence in the framework as well as to advertise capabilities that are contained therein. The following are the expected values and formats of each of the sections in a Discovery message.

- Topic: Discovery
- Metadata: All relevant fields will be populated automatically by the VFAdapter. There are no application specific values in the metadata part of a Discovery message.
- Payload: The payload is expected to be JSON data representing key attributes of the application module and its capabilities. Payload is generated by the enclosed VFAdapter based on information about the application and capabilities that it is able to provide.

The following is an example payload of a Discovery message:

```
{
 "module_uuid":"e9a2a3ea-bb28-11e6-96f7-d4bed98ac412",
 "module_name":"Experiment output from John",
 "module_details":"Conducted on 04-DEC-2016",
 "capabilities":
 [
  {
   "cap_uuid":"09d298dc-ce50-25f9-47a2-e5cde07bd628",
   "cap_name":"PCAP IP distribution at 207 subnet",
   "cap_type_id":"pcap.summary",
   "cap_category":"producer",
   "state":"normal",
   "streaming_data":false,
   "on_demand":true,
   "custom":"{ \"opt1\":\"date\", \"opt2\":\"type\" }"
  },
  {
   "cap_uuid":"12a345bc-ea67-89b0-12c3-d4eab56cd789",
   "cap_name":"DAVC host server d12 cpu usage",
   "cap_type_id":"timeseries.xy",
   "cap_category":"producer",
   "state":"normal",
   "streaming_data":true,
   "on_demand":false,
   "custom":""
  }
 ]
}
```

Table 2 lists discovery message payload fields, and provides a description along with the source of the field value.

Table 2 Discovery message payload fields

Field	Description	Set/generated by
module_uuid	UUID of the application module. Randomly generated.	Framework
module_name	Human-readable name of this application module.	Application
module_details	Optional string data field for additional information about the module. Default is blank.	Application
capabilities	Array of capabilities offered by this application module.	Framework
cap_uuid	UUID of the capability. Randomly generated.	Framework
cap_name	Human-readable name of the capability.	Application
cap_type_id	ID of the capability type, and has specific meaning in this visualization environment.	Application
cap_category	"producer" or "consumer"	Application
state	For future: to indicate the state of the capability; default to be "normal". Other values can be "error".	Application
on_demand	For data producer. True if the capability will only respond after a request is received; false if the capability can generated or broadcast data unprompted. If on_demand is false, the capability is considered in broadcast mode and, therefore, streaming_data are implicitly true. Default is true.	Application
streaming_data	For data producer. True if the capability consists of streaming data. Only meaningful if on_demand is true. streaming_data is assumed to be true if on_demand is false. Default is false.	Application
custom	Optional string data field for additional information about the capability. Default is blank.	Application

Note: UUID = universally unique identifier; ID = identifier.

4.2.1.1 How to Set on_demand and streaming_data Flags of a Capability

Table 3 describes the possible scenarios for using the different combinations of on_demand and streaming_data flags.

Table 3 Scenarios for using on_demand and streaming_data flags

on_demand	streaming_data	Description and use cases
True	False	Capability providing static data, or one-time customized data. For example, content of a csv file in a directory that can drive the visualization of a bar graph.
True	True	Capability that provides real-time data that are customized to a particular data consumer. For example, the capability is a system performance monitoring application, and a data consumer requests only the real-time data of central processing unit load, which is to be published to the data consumer every 2 min.
False	. . .	Capability that provides sporadic or real-time data that are not customized to any particular data consumer and will be published regardless if a data consumer is present. For example, a capability that broadcasts image data every time a camera takes a picture, or a capability that broadcasts the number of logged in user every 30 s.

4.2.2 Request Message

A request message is sent from a data consumer application to a data producer application to request for data. This is required in order to receive data when the `on_demand` flag of the data producer capability is set to true. The request message must state the capability and any appropriate options if necessary.

- Topic: Request: <module_UUID_of_data_producer>.
- Metadata: Origin and msg_id are generated by the framework. correlation_id is not anticipated to be used by a request message.
- Payload: The payload of a request message is in JSON format. The following is an example of payload of a request message.

```
{
  "consumer_uuid":"f0b3b308-ba98-13f6-00f8-c2bf198de692",
  "producer_uuid":"e9a2a3ea-bb28-11e6-96f7-d4bed98ac412",
  "cap_uuid":"09d298dc-ce50-25f9-47a2-e5cde07bd628",
  "req_type":"start"
}
```

Table 4 lists request message fields, and provides a description along with the source of the field value.

Table 4 Request message fields

Field	Description	Generated/set by
consumer_uuid	UUID of the data consumer application module.	Application
producer_uuid	UUID of the data producer application module.	Application
cap_uuid	UUID of the request capability.	Application
req_type	Valid values are "start" and "stop". Only relevant if both on_demand and streaming_data of the capability are true, otherwise blank.	Application

Note: A data consumer is required to request to receive data from a capability whose on_demand flag is false. In this case, the VFAdapter will simply subscribe to the appropriate broadcast message topic. See Data Message Section 4.2.3.

4.2.3 Data Message

Data messages are constructed according to the type of use case that generated the data.

4.2.3.1 On-Demand Data Message

An on-demand data message is sent by a data producer application module to carry data that have been requested by a data consumer module. In this case the `on_demand` flag of the data producing capability is true.

- Topic: OnDemandData.<module_UUID_of_data_consumer>.<msg_id_of_request_msg>
- Metadata: Origin and msg_id are generated by the framework.
  - correlation_id: set by the data producer application based on the msg_id of the request message that it had previously received. correlation_id should be identical to the msg_id_of_request_msg of the topic.
- Payload: Content of the data. The data are prepared by the data producer application, and they can be string or binary data.

4.2.3.2 Broadcast Data Message

Additionally, a data producer module may use a data message to broadcast data associated with a capability that does not require requests. The data are broadcast regardless of whether an active consumer exists. The `on_demand` flag of the data producing capability is false.

- Topic: BroadcastData.<module_UUID_of_data_producer>.<producer_cap_uuid>
- Metadata: Origin and msg_id are generated by the framework. correlation_id is not used.
- Payload: Content of the data. The data are prepared by the data producer application, and they can be string or binary data.

4.2.3.3 Directed Data Message

A directed data message is pushed by a data producer application to a specific data consumer application. In the case of directed data, the data producer controls the flow, while the data consumer is a passive recipient of the data. This is in contrast to on-demand data.

- Topic: DirectedData.<module_UUID_of_data_consumer>.<consumer_cap_uuid>
- Metadata: Origin and msg_id are generated by the framework.
- Payload: Content of the data. The data are prepared by the data producer application, and they can be string or binary data.

5. Application Work Flows

The following are some of the work flows that are typical for ARLVF applications.

5.1 Startup

- Application initializes and then instantiates the VFAdapter and provides to it information such as application module name and IP and port numbers to connect to the Executive Controller.
- Data producer application registers its own Capabilities with the VFAdapter.
- Data consumer application specifies compatible Capability filters with the VFAdapter. The filter allows the data consumer to focus in on those data producers and capabilities that are compatible with this data consumer. If not specified, the data consumer application will see all discovered applications and their registered capabilities.

- The VFAdapter subscribes to the following message topics, which will remain active throughout the lifetime of the application.
    - Discovery
    - Request.<module_UUID_of_application>
    - OnDemandData.<module_UUID_application>
    - DirectedData.<module_UUID_of_application>

5.2 Discovery

The Discovery mechanism is handled solely by the VFAdapter object contained in the ARLVF application.

5.2.2 Outgoing

On a regular basis, the VFAdapter scans the registered Capabilities of the application module. It then constructs and sends the Discovery message to the Executive Controller, which in turn publishes to all application modules in the ARLVF environment. This functionality executes automatically without the intervention of the application module.

5.2.2 Incoming

As a result of the subscription to the Discovery topic, the VFAdapter will be notified by ZeroMQ when a discovery message is received. Incoming discovery messages are digested by the VFAdapter, which in turn constructs/updates the internal data model representation of the peer application module and its Capabilities. By implementing the appropriate callback functions, the enclosing ARLVF application can be notified when a new application is discovered.

When the VFAdapter does not receive a discovery message from a peer application after a certain amount of time, the internal representation of that application module is removed from the list of discovered modules. The enclosing application can be notified of such event by implementing the appropriate callback function to perform additional handling.

Additionally, the enclosing application has access to the data model at any given time during runtime.

5.3 Synchronous Request and Response of On-Demand Nonstreaming Data

The following describes the sequences of events that occur when a data consumer requests for and receives data from a data producer.

1) The ARLVF data consumer application invokes the **`request_data`** function. The message includes information about the data producer and the capability that provides the desired data. Additional query options can be included in the request message.

2) The VFAdapter publishes the request message. It also puts away the message_id of the request message in an internal cache so it can be cross referenced when the corresponding response data arrives. At this point, the execution of the **`request_data`** function is blocked, awaiting response data. There is a timeout associated with this block, which defaults to 30 s. The timeout value is configurable as a parameter in the **`request_data`** function call.

3) The Executive Controller receives the Request message and forwards it to the appropriate ARLVF data producer application.

4) By virtue of the subscription to the **`Request.<module_UUID_of_application>`** message topic, the VFAdapter of the data producer application is notified by ZeroMQ about the incoming request.

5) The VFAdapter invokes the callback handler function that the data producer application has declared.

6) The data producer application looks up the requested capability and associated options and prepares the data accordingly. It then invokes the send data function of the VFAdapter to publish the on-demand data. The correlation_id of the Data message must be correctly set.

7) The Data message reaches the Executive Controller, which immediately publishes it to the appropriate data consumer.

8) Because it already subscribes to the **`OnDemandData.<module_UUID_application>`** topic, the VFAdapter on the data consumer application receives the Data message. It finds the correct message_id and releases the lock on the execution of the **`request_data`** function.

9) At the data consumer application, the `request_data` function returns with the Data message created by the data producer application.

Figure 2 illustrates the messaging and processing of the request.

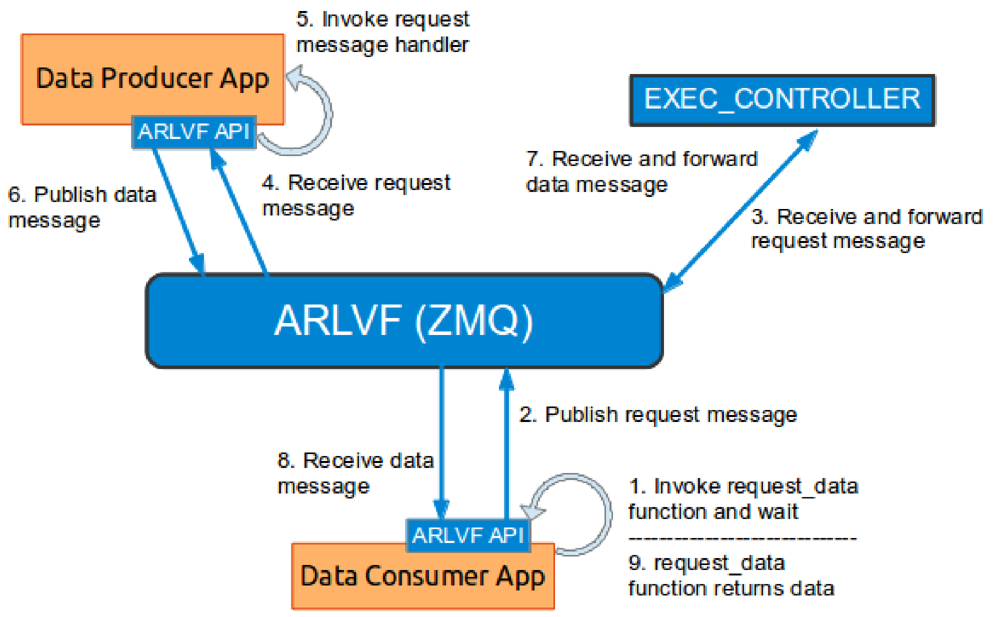
Data Producer App
ARLVF API
5. Invoke request message handler
EXEC_CONTROLLER
7. Receive and forward data message
3. Receive and forward request message
4. Receive request message
6. Publish data message
ARLVF (ZMQ)
2. Publish request message
8. Receive data message
1. Invoke request_data function and wait
9. request_data function returns data
ARLVF API
Data Consumer App

Fig. 2 Synchronous request and response

6. Conclusion and Roadmap

The ARLVF provides a platform for reusable scientific visualizations to researchers working in the ARL Network Science Research Laboratory, thereby reducing the researcher's workload. Visualizations already implemented in ARLVF range from simple graphs, strip charts, and bar charts to complex network node mapping, node interaction, and packet analysis. ARL developers created VFAdapters using Python, Java, JavaScript, and Node-RED to connect to previously developed experiments and research tools and expanding the utility of the ARLVF.

The ARLVF will continue to grow as developers and researchers encounter additional opportunities and use cases. The following are potential features for future implementation.

- Command and control of application modules. Currently, data consumers initiate data requests. There will be cases in the future where such requests would come from a different application. Additional message types and handlers will be required to support this feature.

- Status and Statistics reporting. Create a uniform mechanism to collect and report metrics of each application module.
- Additional language support. The VFAdapter and the rest of the ARLVF API are written in Python. It may be desirable to expand to other programming languages such as Java.

7. References

1. Dron W, Keaton M, Hancock, J, Aguirre, M, Toth AJ. US Army Research Laboratory visualization framework design document. Adelphi Laboratory Center (MD): Army Research Laboratory (US): 2016 Jan. Report No.: ARL-TR-7561.

2. US Army Research Laboratory. The Network Science Research Laboratory. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Sep 14 [accessed 2017]. http://www.arl.army.mil/nsrl.

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
ARLVF	US Army Research Laboratory Visualization Framework
API	application programming interface
ID	identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
UUID	universally unique identifier

1	DEFENSE TECHNICAL
(PDF)	INFORMATION CTR
	DTIC OCA
2	DIR ARL
(PDF)	IMAL HRA
	RECORDS MGMT
	RDRL DCL
	TECH LIB
1	GOVT PRINTG OFC
(PDF)	A MALHOTRA
2	ARL
(PDF)	RDRL CIN T
	C HSIEH
	A TOTH

INTENTIONALLY LEFT BLANK.