Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Datateknik
2022 | LIU-IDA/LITH-EX-G--22/030--SE

Temporal Sparse Encoding and Decoding of Arrays in Systems Based on the High Level Architecture Standard

Kodning av stora datamängder i system baserde på High Level Architecture standarden

Johan Thörnblom
Viktor Severinsson

Supervisor : Pontus Haglund
Examiner : Filip Strömbäck

External supervisor : Glenn Wissing

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

In this thesis, a method for encoding and decoding arrays in systems based on the standard High Level Architecture is presented. High Level Architecture is a standard in the simulation industry, which enables interoperability between different simulation systems. When simulations share specific data with other simulations, they always send all parts of the data. This can become quite inefficient when the data is of an array type and only one or a few of its elements' values have changed. The whole array is always transmitted regardless whether the other simulations in the system need all elements or just the ones that have been modified since the last transmission. Therefore there might be more traffic on the network than needed in these cases.

The proposed method, named Temporal Sparse Encoding, only encodes the modified elements when it needs to, plus some additional bytes as overhead, that allows for only sending updated elements. The method is based on the concept of sparse arrays and matrices, and is inspired by the Coordinate format, which uses extra arrays with indices referring to specific elements of interest.

In a small simulation system, acting as a testing environment, it is shown how Temporal Sparse Encoding can save both time and above all, bandwidth, when sharing updates. Each test was carried out 10 times and in each test case 1 000 updates were transmitted. In each test case the transmission time was measured and the compression ratio was calculated by dividing the number of bytes in the encoding containing all elements by number of bytes in the encoding containing just the updated ones.

The biggest compression ratio was calculated to be 750.13 and came from when 1 out of 1 000 elements were updated and transmitted. The smallest compression ratio was 1.00 and came from all the cases where all the array's elements were updated and transmitted. Some of the conclusions that were made was that the Temporal Sparse Encoding can save up to 33% of the time compared to the standard encoding and that a lot of the transmission time is spent on extracting elements once they have been decoded. These findings suggest that endeavors in optimization should be focused at the language level, specifically on management of data, rather than the transmission of data when there is not a lot of traffic occurring on the network.

Acknowledgments

We want to thank the company Pitch Technologies and their employers for providing this thesis work and supporting us. We would also like to thank our supervisor at Pitch Technologies, Glenn Wissing, for providing a lot of support and helping us make this thesis possible. We thank our supervisor Pontus Haglund and our examiner Filip Strömbäck at Linköping University for their assistance in making this thesis academic.

Contents

List of Figures

List of Tables

1 Introduction

The first section of this chapter, the background, information is presented about the company that this thesis project was carried out at. The second describes what this thesis is about, namely the problem of sending a lot of data in systems based on the High Level Architecture (HLA) standard. The third section, the aim, describes what was tried to be achieved, which concisely was better performance. The fourth one contains the research questions and the last one describes what assumptions and delimitations that were used.

1.1 Background

Pitch Technologies is a company that develops distributed systems for simulation. Their products support the standards HLA IEEE 1516-2010 (also known as HLA Evolved), HLA IEEE 1516-2000, and HLA 1.3. Pitch Technologies offers a product called *Pitch Talk*™, which is a radio simulator. *Pitch Talk*™ allows you to customize effects on the radio signal, such as degradation effects due to poor signal as a consequence of distance or transmission blocking obstacles. With many radio devices present in a system of simulations the amount of data can easily become large, as the possible connections between radio devices increases.

1.2 Motivation

The amount of data sent between different systems increases in conjunction with the growth of technical development and thus the importance of quick data transfer increases. In distributed systems data exchange occurs frequently, therefore it is important to minimize the amount of data to exchange in order to save time and resources.

Many distributed systems used for simulation are implemented according to the standard HLA IEEE 1516-2010 [1]. These systems are primarily used to combine several simulation systems known as federates, which together form a group of systems known as a federation [2]. HLA is widely used in the defence sector, but is increasingly used in the civil sector [2].

The lack of some sort of predefined sparse encoding for the predefined array data types *HLAfixedArray* and *HLAvariableArray* has been brought up in the past [3] with regards to the new HLA standard successor to IEEE 1516-2010, usually referred to as IEEE 1516-202x or HLA 4.

In a federation, different federates can own and contribute with their own data, selected from a shared object model definition. Depending on the system, different federates need to know more or less about other federates' data and its current states. The extent of this knowledge is decided though a *publish/subscribe scheme* [4]. With this scheme, every federate decides which data it is going to share by "publishing" it, and then the other federates can decide which of the published data they want the latest states of by "subscribing" to them [4].

When a federate in an HLA system updates its data that other federates subscribe to, the subscribing federates need to be notified and receive the updated version of the data. Thereby the federate has to send data to all of the other federates that are interested in that specific data. The data traffic then grows rapidly as the number of federates increases in a federation. To keep data traffic and the transmission time minimal, it is important to send as little redundant information as possible [5].

When a federate has data that is published, that other federates subscribe to, there are sometimes parts of that data that do not provide the receiver with any new information. In this thesis, this data is referred to as *redundant information*. It would be identical parts of attribute values that the receiver already has. If for example there is an attribute in the form of a record of 10 different elements and only one of them gets updated before a transmission, then the whole attribute will be transmitted by default. This redundant information is transmitted because there is no method specified in the HLA standard of filtering it out before all parts of the data is sent. The bigger the data of an attribute being sent within an HLA-based system, the bigger the problem with redundant information. Especially when only a small fraction of the data set is interesting.

This thesis examines how it would be possible to minimize the redundant data being transmitted in an HLA-based system while maintaining the correctness of the relevant data and the effects of it. It is examined how the concept of sparse encoding, that is encoding of sparse arrays or sparse matrices, can enable this. The effects of minimizing redundant data, when it is profitable to do so and when it is not are also examined.

1.3 Aim

The aim of this thesis is to investigate how much one can minimize the redundant data exchanged between HLA federates when they exchange large arrays of data, and to what degree this can be done on top of the HLA standard. In this thesis, a proposed way of doing that is investigated by looking at what additional information that needs to be sent, how to encode and decode array content effectively and to what degree the HLA standard supports such a solution.

This investigation was done through the implementation of a method. Due to the method's similarities to Sparse Encoding, we call this method Temporal Sparse Encoding (TSE). Sparse encoding techniques are used when most elements in an array are of value zero or null [6], but for this thesis the TSE was more suitable because of its time aspect. Rather than encoding based on whether elements have a zero/null value or not, it is based on whether elements have changed over time or not. Thus the aim is also to investigate how well TSE can achieve the desired effects described in the previous paragraph.

To evaluate the effectiveness of the TSE, transmission time, transmission size and transmission load was measured. The transmission size (the size in bytes of a transmitted update) was measured in order to be able to calculate compression ratio. Transmission time (the time it takes to encode, transmit, receive and decode an update) was measured in order to be able to evaluate the overall improvement or deterioration of the data sharing process. Transmission load is the transmission size divided by the transmission time, and is calculated to give an indication of network strain.

1.4 Research Questions

When encoding and decoding data in large arrays that are to be sent within a system based on the HLA standard:

RQ1 What are properties of the array, for which the proposed TSE technique may contribute to shorter transmission time and smaller transmission size?

RQ2 How does the transmission time and transmission size differ when using the proposed TSE technique compared to when using the standard technique described in HLA?

RQ3 How does the transmission load differ when using the proposed TSE technique compared to when using the standard technique described in HLA?

1.5 Delimitations

In this section the delimitations of this thesis are presented.

1.5.1 Transport-Layer Protocol

HLA supports network communication through both TCP and UDP. To be able to assume that the transmitted data always reaches its destination, the updates were solely transmitted through the protocol TCP and not through UDP. Thus, the findings of this study may not be applicable to systems relying on transmissions through the protocol UDP.

1.5.2 Data Type Encoded

In this thesis, the possibilities of encoding and decoding variable arrays sparsely will be examined. The elements of the array will be seen and handled as indivisible objects and possible effects of making them sparse will not be examined. The elements must also be of an HLA data type.

1.5.3 Number of Federates Connected

The majority of the tests were carried out with two federates connected to the federation execution. Tests with four federates was also carried out to be able to examine how the performance change when more federates connect. The number of federates connected in this thesis was delimited to two and four since having more federates would require more computers. Moreover, comparing the results produced with two federates connected compared to with twice as many connected would presumably illuminate whether adding more federates would make TSE more preferable or not anyway.

2 Theory

In this chapter, the concepts of HLA and sparse arrays and matrices are explained.

2.1 High Level Architecture

The purpose of HLA is to specify a common technical architecture for the backbone of interconnected simulation systems [1]. HLA IEEE 1516-2010 is a standard for these types of systems that are used for example by the US Department of Defense, NATO and NASA. An HLA-based system consists of *federates*, which together form a *federation*.

2.1.1 Federates

Federates are the simulations themselves. They can be computer simulations, manned simulations or other simulation systems. A requirement for federates is that they need to implement certain functionalities so that their objects are able to interact with objects that exist at other federates [1]. This should be handled though data exchanges by services implemented in the *Run-time Infrastructure (RTI)*. RTI is a middleware that enables the communication between federates by providing them with adequate services [1]. The relation between the RTI and connected federates can be seen in Figure 2.1.

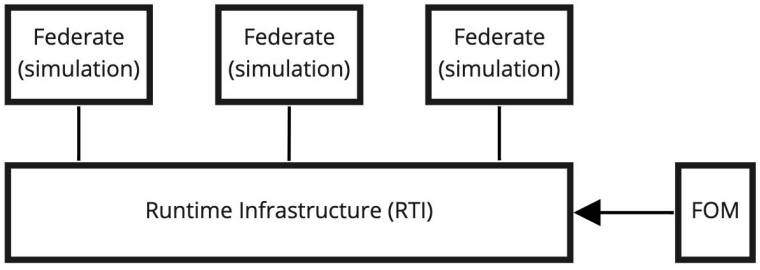

Figure 2.1: Topology of a typical interconnected system based on the HLA standard.

2.1.2 Federation Object Model

The Federation Object Model (FOM) describes how federates communicate, thereby, it is also included in Figure 2.1. The FOM evokes RTI services so that interaction and updates may be shared among the federates and provides a standard way for federates to send responses to the RTI and to interact with it [1]. It also specifies what is sent in the federation during the execution of it and consists of objects, object attributes, interaction classes, interaction parameters and specified data types [7].

2.1.3 Federation

Inter-connected simulation systems of federates, together form what is called a federation. The execution of a federation is called a *federation execution*. A federation consists of three main components which are federates, an RTI and a FOM [8]. The first federate that starts running is the one that creates the federation execution. The federates that start running after that will notice that a federation is already executing and thus they will just join the execution instead of creating a new one.

On an interface level, the federation gives the appearance of being built upon an infrastructure in the shape of a service bus [7]. This means that each federate only needs to have one connection within the federation and that is to the RTI. This service bus shape is also demonstrated in Figure 2.1. The fact that all federates are interpreted to be connected directly to RTI, means that if a federate is added, removed or replaced, none of the other federates need to be updated, as the federation's RTI takes care of this. In actuality the structure is like a pair-wise integration, which means that a federate needs to establish one connection to each federate that it intend to exchange services with. [7]. This is however taken care of in a way so that the real structure is not noticed by the federates.

2.1.4 API

Two important interfaces from the HLA standard are the *RTIambassador* and the *FederateAmbassador*. The *RTIambassador* is an interface of the RTI with the purpose of presenting itself to the federate, and the *FederateAmbassador* is an interface on the federate side that communicates with the *RTIambassador* and needs to be implemented by the federate [4]. The two classes are necessary in order for the communication in a federation to work.

HLA has some pre-defined data types. Examples of these are *HLAinteger64BE* (BE as in Big Endian), *HLAfloat64BE*, *HLAfixedRecord*, *HLAfixedArray* and *HLAvariableArray*, which all inherit from the HLA base type *DataElement*. Each of these data types has its corresponding encoding helper class of the same name. Usually these are referenced as data types in the context of FOM and encoder helper classes when used in programming code. In the C++ implementation there is a class for each data type which represents the data type and contains its encoder. The name of the encoding helper class is in this thesis used to refer to these types. The term **decoder** is used to refer to an encoding helper class that is only used for decoding.

The *VariableLengthData* interface, which is part of the C++ API, is the data type in which encoded data is stored right before it is being sent. It holds an array of bytes and can be used to transmit data between federates [9].

Federates can use the encoders to encode their data in order to share it to other federates in the federation. With *HLAfixedArray* and *HLAvariableArray*, updates always contain all elements when shared within the federation [9], even if only a few of the elements actually have changed. If for example an array that contains 100 elements is shared with other federates and only two or three elements has changed, there is no current HLA data type or functionality that enables the sending federate to only send a packet containing the updated elements (or a superset thereof).

2.1.5 The Request/Provide Exchange Scheme

It is not necessary for federates to wait for updates of instances they subscribe to. They may also request them by using the request/provide exchange scheme described in the HLA standard [4]. The `requestAttributeValueUpdate()` is a function which a federate can call to request the latest update of an attribute instance. When this function is called, the RTI will call the function `provideAttributeValueUpdate()` at the federate that owns the attribute instance asked for. In that function, functionality for sending the data asked for is usually implemented.

2.2 Encoding and Decoding

The HLA standard also specifies certain criteria for the HLA data types and how they should be encoded and decoded. The standard also specify an encoding package, which provides *encoding helper classes* for each HLA data type. These classes must support encode and decode methods in order to convert a data type to its corresponding byte-array and vice versa. [4]

2.2.1 HLAvariableArray

HLAvariableArray is the pre-defined encoder helper class (and data type) for dynamic arrays in HLA [4]. The interface of *HLAvariableArray* provides methods for adding elements of HLA data types, and to set the value of a chosen element. The *HLAvariableArray* can be configured to either own and store its element by using the functions `set()` and `addElement()`, or to point to already existing elements by using `setElementPointer()` and `addElementPointer()`.

While the interface of the *HLAvariableArray* has to conform to the HLA standard [4], the language specific implementation of the encoder class is not specified. Although according to the Object Model Template (OMT) specification of the HLA standard, the implementation for encoding must follow the rules of padding and alignment [9].

The encoding of the *HLAvariableArray* starts with four bytes containing the number of elements encoded as a 32-bit integer, or more specifically an *HLAinteger32BE*, as per requirement of the OMT specification [9]. Thereafter, the encoded bytes of each element follow. The data structure is demonstrated in Table 2.1.

Table 2.1: Structure of encoded data when using the *HLAvariableArray*, padding not displayed.

Field size (bytes)	Field name	Data Type
4	number of elements	Integer32
X	element 0 payload	<data type>
X	element 1 payload	<data type>
...	...	...
X	element n payload	<data type>

The decoding of *HLAvariableArray* is fairly similar to the encoding. The first 4 bytes representing the size of the array is first decoded, followed by all elements that has been transmitted. Each of these elements decode itself from its respective chunk of bytes.

2.3 Sparse Array/Matrix

A sparse array, or a sparse matrix, is a container where most elements are of value zero or null [6]. The opposite is a container with elements that primarily consists of non-zero values, which is known as a *dense* array or matrix [5, 6].

There are different methods of compressing a sparse array or matrix. In this section the Coordinate format, which is a format the proposed technique of this thesis draws inspiration from, is presented. In chapter 3, more methods for handling sparse arrays and matrices are presented.

2.3.1 Coordinate format

One of the most basic methods of storage is the Coordinate (COO) format [10]. It maps the non-zero elements of a sparse matrix into three arrays representing the value of the element, the row index and column index. The advantages of COO is that it is simple and flexible, and it demonstrates the idea of a sparse array/matrix.

An example scenario when the COO format is used is demonstrated in Figure 2.2. The raw data, the actual matrix, is to to the left in the figure and its compressed version is to the right. The first of the resulting arrays contains all the non-zero values of the original matrix and the second and third contains coordinates, or indices, of the positions in the original matrix where the corresponding values are supposed to be located.

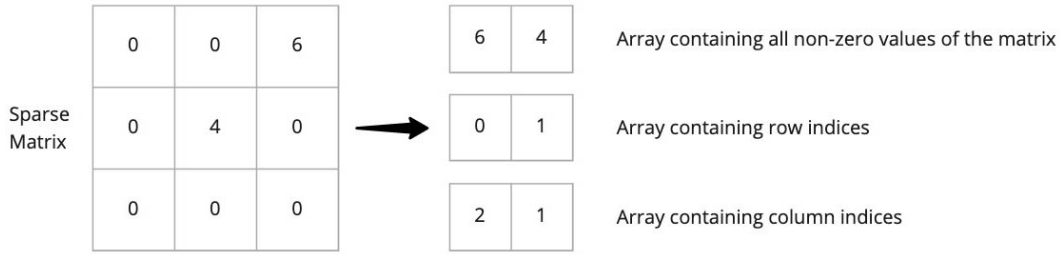

Figure 2.2: An example scenario of how a matrix is compressed using the COO format.

3 Related Work

There is not much previous research done about TSE for HLA-based systems. There are however more research done about sparse encoding, especially the sparse array/matrix methodology that describes, among other things, different methods for representing data in sparse arrays or matrices [5, 6, 11]. Yan et al. [6] also examines how sparse arrays/matrices can be made efficient on modern computers with multi-core systems. The methods presented in this chapter aim to improve the already existing COO.

3.1 Compressed Row Storage

Compressed Row Storage (CRS), or sometimes referred to as Compressed Sparse Row (CSR), is the most common method for sparse matrices, and just like COO it omits the zero value elements of a matrix and keeps track of the non-zero values through three arrays [10]. The idea is similar to the one of the COO format, but instead of storing the row and column for each non-zero value, the non-zero values and their columns are stored for each row. Therefore it has an array of column indices and an array with non-zero values, just like COO. The third array represents the matrix's rows. For example the *i:th* element of the row-array represents the *i:th* row in the matrix, the element then points to the first non-zero value and its column belonging to the row.

Thus the representation for the rows is compressed to the number of rows for a given matrix. Making it a more efficient method than COO in terms of space as a matrix grows larger. However in cases when when the row count is one and the matrix actually is an array, the CRS methodology becomes more like the one of the COO. Since there in these cases only is one row a non-zero value can belong to, there is essentially no need for a row-array or reason for further compression.

Because this method often shrinks the size of sparse array compared to COO, it is often preferred and generally more efficient when performing computations. There is also a variation to this scheme called Compressed Column Storage (CCS), it works just like CRS but the compression is done for the column rather than the row [10].

3.2 Dynamic Compressed Sparse Row

King et al. [12] mention the lack of ability to dynamically update sparse matrices with the current storage methods, such as CRS. They propose a format which they call Dynamic Compressed Sparse Row (DCSR). DCSR similarly to CRS has a row pointer array, however DCSR additionally stores segment offsets to allow for future insertions. In their scheme, King et al. [12] also stores a row size to support *Adaptive CRS* and other optimization techniques, although this requires customized kernels and thus are not in the scope of this thesis.

This method might be worth considering when dealing with dynamic arrays as DCSR might allow for dynamically changing the sparse array without having to re-encode after each modification of the array.

3.3 Extended Karnaugh Map Representation-Compressed Row/Column Storage (ECRS/ECCS)

Lin et al. [11] proposes methods for sparse encoding of multidimensional arrays. They claim that the most common methods for compressing data in a sparse array such as CRS/CCS schemes are not efficient enough for multidimensional arrays. They present two methods for compressing data, ECRS and ECCS, which are analyzed and according to them proven more effective than the traditional methods.

The ECRS/ECCS schemes are based on the Extended Karnaugh Map Representation (EKMR) scheme. This algorithm represent multi-dimensional arrays as sets of two-dimensional arrays in order to make matrix operations between them simpler [13].

The optimization the ECRS/ECCS schemes have, compare to what the CRS/CCS schemes have, is that they do not need an extra array of index for every dimension in the sparse matrix. Instead they interpret the multi-dimensional array as if it had a less amount of dimensions, and therefore, more indices can be stored in the same index array [11] .

3.4 Ellpack

In the Ellpack format the number of rows is dynamic but the number of columns is fixed. The format can be memory inefficient since the size of the rows is fixed, but on the other hand useful in the cases where the matrix to be compressed also has a fixed number of columns. Because of its fixed row size, insert operations are usually fast with the Ellpack format since the space needed for the new entries often already is allocated [12].

3.5 Data Distribution Schemes

Usually, the method for encoding data to a sparse array consists of three phases: data partition, data distribution, and data compression. Send Followed Compress (SFC) is a scheme that is divided into these three phases in the aforementioned order. In their work, Lin et al. [5] propose two new schemes that differ from SFC which they call Compress Followed Send (CFS) and Encoding-Decoding (ED). In CFS the compressing is done before the distributing and in ED the distribution phase is divided into one encoding part and one decoding part that takes place both before and after the distribution phase.

Lin et al. [5] compared their schemes with the SFC scheme, both through theoretical analysis and through experimental tests, and concluded that their algorithms surpassed SFC in most contexts. Mutually, ED outperformed CFS in all tests.

4 Method

To achieve the goal of this thesis, which was to shorten transmission time and to minimize data traffic within in HLA-based systems, the idea was to only transmit the parts of the data that had been modified rather than all of its parts. To a certain extent, the implementation of this idea resembled the technique of spare encoding, commonly used for handling arrays or matrices with most of its elements set to zero or null [6]. The difference was that instead of distinguishing elements depending on whether they had the value zero or not, they where distinguished depending on whether they had changed or not since the last transmission took place. Because of this temporal aspect of the elements, the technique used in this thesis was labeled *temporal sparse encoding*.

4.1 Proposed Encoder/Decoder: SparseVariableArray

The TSE technique was implemented in a new encoding helper class called *SparseVariableArray*. The *SparseVariableArray* can be instantiated as an object instance, either as an encoder or as a decoder and is based on the *HLAvariableArray* with the same interface. Just like the *HLAvariableArray*, the *SparseVariableArray* can be used for dynamic arrays. Elements can be added to the object instance by calling the method `addElement()` and its already existing elements can be updated by calling the method `set()`. It is also possible to remove any element from the object instance and to insert an element at any position. These function calls are the first step in the transmission process which is displayed in Figure 4.1.

When the federate is done altering its elements in the encoder, the method `encode()` should then be called to prepare the data for being transmitted. The `encode()` method, just like the *HLAvariableArray* counterpart, returns a `VariableLengthData` object with the

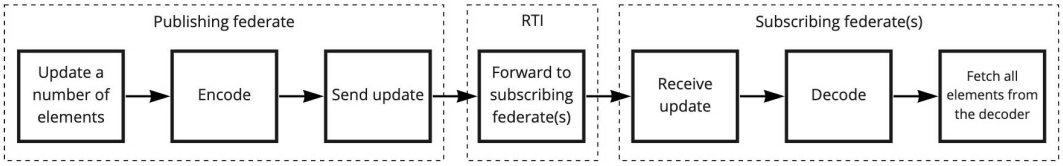

Figure 4.1: The process of transmitting updates when using the *SparseVariableArray*.

encoded bytes. These are then transmitted to the RTI which forwards it to the subscribing federates. At a receiver's side, when a `VariableLengthData` object arrives from the RTI, the `decode()` method should be called on the *SparseVariableArray* decoder object and input the `VariableLengthData` object as a parameter to decode it. Then the decoded data should be fetched from the decoder.

The *SparseVariableArray* can be used in the same way as the *HLAvariableArray* except for that the methods `setElementPointer()` and `addElementPointer()` are not available.

Internally the *SparseVariableArray* encoder/decoder stores a state of the actual data. In that way it can detect and handle incoming changes from the federate that owns the attribute instance. Because of this, each *SparseVariableArray* instance should only be associated to one specific attribute instance to maintain the consistency. In other words, every local array instance that a federate owns needs to have its corresponding encoder/decoder for itself. When an element in the encoder is modified, the index to that element is inserted into a dictionary inside the encoder and is kept there until it is time to transmit the elements.

Sometimes it is feasible to send all the information of the attribute instance and not just the updated parts. An example of when that would be the case is when a new federate joins the federation execution and makes a subscription request. At that point it will not know any information of the attribute instance it intends to subscribe to. Another example is when a federate already is connected but requests a full update of the attribute anyway. When requests like these are made, it is important to respond with every element of the attribute instance and not only the recent updates to maintain consistency.

The *SparseVariableArray* solves this problem by utilizing the request/provide exchange scheme from the HLA standard. In the function `provideAttributeValueUpdate()` it is suitable to prepare the *SparseVariableArray* for sending all elements. In that way, the subscribing federate can get all elements by simply calling the function `requestAttributeValueUpdate()`. When a new federate joins the federation execution, the RTI calls the function `discoverObjectInstance()` at all connected federates for all registered attribute instances. In that way the new federate will get the full updates from all relevant federates.

In some cases however this can create another problem. If a new federate connects that does not have any version of a specific shared array, and if it before it receives the whole array, receives an update containing only a fraction of the elements with indices referring to elements which the federate does not yet possess, errors will occur if the situation not is handled. The *SparseVariableArray* handles this by checking whether it has received any full update of the array before or not, and if it has not, it will ignore the update and wait for the full one.

A way for the sending federate to send all elements is by using the *SparseVariableArray*'s `set()` method for each and every element in the local array instance, and after that, encode and send the whole array. In that way, the encoder will know that all elements have changed and therefore send all of them to the subscriber, without the extra overhead needed for sending it sparsely.

There was a choice that had to be made whether the sparse array functionality should be implemented in an encoder/decoder class or directly in the federate API. At the end it was implement in an encoder/decoder class so that it would be possible to apply the functionality in any HLA-based system and not just in the tests environment of this thesis.

4.2 The SparseVariableArray Algorithm

The *SparseVariableArray* was, like the *HLAvariableArray*, implemented with an internal dynamic array to point to all HLA data type versions of the elements. With the *HLAvariableArray* encoder/decoder, these elements could either be owned by the encoder or exist outside

of the encoder. At the point in time when this thesis was carried out only the functionality in the *SparseVariableArray* of handling owned elements was implemented and examined.

4.2.1 Detecting Updates

For the encoder to know which elements that had been updated, two methods were considered. The first method implied looping through the internal dynamic array (the list of elements) of the encoder and compare each element with the corresponding local element in the federate API and then update those elements in the internal dynamic array of the encoder that differed. This would be done whenever the encoder encoded its elements, since it at that point would need to know what had changed. The second method, which was used in this thesis, was to keep track of all indices of the elements in the array that was updated through the API.

The data structure of the dictionary inside the *SparseVariableArray* where the indices referring to the modified elements are stored is a search tree. The keys in this dictionary are all unique. With this dictionary it is possible to map the indices to the internal container and thereby have a sparse representation in the shape of a COO format of the element that is to be transmitted. The set of indices is updated every time the API uses the methods `set()` or `addElement()`.

To be able to detect updates it is necessary to compare two different states of an element. The local version of each element at the API always contains the current state while the old state is stored inside the decoder.

4.2.2 Encoding

The *SparseVariableArray* can encode its elements in two ways, either sparsely or densely, it would for example encode a dense array by encoding all of its elements. The *HLAvariableArray* functions that the *SparseVariableArray* uses modified versions of are `getEncodedLength()` and `encodeInto()`.

Every time the `encode()` method is called, the encoder first checks whether all elements have been modified or not. It does this by checking whether the total number of elements in the array is the same as the number of elements in the dictionary containing indices or not. If it is the same, the encoder will start encoding densely immediately and if it is not, the encoder will start calculating which approach to use by examining which method that would produce the data with the fewest number of bytes intended for transmission. The public method `getEncodedLength()` from the *HLAvariableArray* is used to calculate the expected length of sending the data densely and another modified version of it is used to calculate the length when sending the data sparsely. The results of these calculations are then compared in order to decide which type of encoding to use.

An alternative way to decide whether to encode sparsely or not would be to look at the number of elements changed versus the total number of elements, or more specifically the ratio between the size of the array of indices and the array of all elements. If the ratio then would be big enough, the encoder would switch to encoding densely. It would however be problematic to set this ratio to a big value in some cases. Such a case would be when the elements are small in size, since the proportion to the size of their corresponding index data would be relatively big. An example would be if an array with 32-bit integers as elements would be encoded sparsely. Then already when half of the elements are updated, the sparse encoding would become as large as encoding would be when encoding densely.

To indicate which type of encoding to use, a Boolean data member is used as a flag in the *SparseVariableArray* class. The *SparseVariableArray* also provides a method for always encoding non-sparse. Such a method is useful when all parts of the data needs to be sent, like for example when a new federate joins the federation execution, or when the performance

difference of dense encoding between the *SparseVariableArray* and the *HLAvariableArray* is compared.

Compared to the *HLAvariableArray* encoding, which can be observed in subsection 2.2.1, the data structure of encoded data of the *SparseVariableArray* differs, as shown in Table 4.1. The first difference is that it starts with four bytes representing an integer flag that indicates whether the data is encoded sparsely or densely. The flag is represented as integer data type and it is set to one if the data is sparse or zero if it is not. This is because the encoded data is different when encoding densely, and thus it needs to be decoded differently. The dense encoding code is the same as the original encoding code, apart from the sparse flag, to maintain the same performance.

Field size (bytes)	Field name	Data Type
4	isSparse flag	Integer32
4	number of elements	Integer32
X	element 0 payload	<data type>
X	element 1 payload	<data type>
...	...	...
X	element n payload	<data type>

Table 4.1: Structure of encoded data when using the *HLAvariableArray*, padding not displayed.

In the case where the data is encoded sparsely, which is depicted in Table 4.2, the sparse flag is set to one and then the four bytes representing the number of elements follow as an integer data type. Then for every element, a four byte integer is prepended indicating which index in the original array that it refers to.

Field size (bytes)	Field name	Data Type
4	isSparse flag	Integer32
4	number of elements	Integer32
4	number of elements updated	Integer32
4	element 0 index	Integer32
X	element 0 payload	<data type>
4	element 1 index	Integer32
X	element 1 payload	<data type>
...	...	...
...	...	...
4	element n index	Integer32
X	element n payload	<data type>

Table 4.2: Structure of encoded data when using the *SparseVariableArray*, padding not displayed.

A 32-bit integer is used as size for the indices for every element, since that is the size *HLAvariableArray* is specified to encode [9]. The encoded data that the *SparseVariableArray* encodes uses the COO format (with 1 row) since the array is interpreted as one dimensional.

4.2.3 Decoding

The first step in the decoding process is to read the first four bytes, which represent the flag indicating whether the data should be interpreted as sparse or dense. If it should not, the rest of the encoding process is the same as for the *HLAvariableArray*. If it should however, the next component that is to be decoded is the array size, which then is used to resize the decoder's internal element list. The step after that is to decode the encoded array size indicating the number of elements needed to be decoded. The original size is required, because sometimes it is necessary to add or remove elements, and in those cases the *SparseVariableArray* needs to resize itself to be able to hold the additional elements. Finally, the associated index for each element is decoded together with its corresponding element which decodes itself until all elements are decoded.

Because elements are stored within the *SparseVariableArray*, it simply updates its internal container's elements with the new elements from the decoded data. Therefore it is dependent on having received all data beforehand for a sparse decoding of an update to make sense.

4.3 Testing Application: MiniTalk

To simulate real world cases a C++14 application called *MiniTalk* was built which was a very simplified imitator of Pitch Technologies' product *Pitch Talk*™. *Pitch Talk*™ is a radio simulator in which participants can talk to each other and where the sound sent between radios can be altered with different sound effects.

By executing the code in *MiniTalk* one federate would start running and create a federation execution. It would then be possible to start several federates from that particular code which then would join the federation initialized by the first federate. Thus, *MiniTalk* can run on different computers and each computer can run either one or more instance of *MiniTalk*.

MiniTalk uses a class that was constructed called *Participant*, which in the FOM was represented as an object class. To represent all joined federates, each federate would have one *Participant* object instance per joined federate, even one representing itself. The *Participant* class contained data members like encoders, decoders and containers that are represented in the FOM as attributes.

Each federate in the federation owned and contributed with a specific sequence container of radio units which the other federates subscribed to. In the FOM this container was represented as an attribute. The container of radio unit records were stored in a federate class which inherited from the federate ambassador class from the HLA standard. A radio unit was depicted as a record of ID, latitude position, longitude position and a description in the form of a string. The corresponding HLA data types for the radio unit would be an *HLAfixedRecord* with an *HLAinteger64BE*, two *HLAfloat64BE* and an *HLAunicodeString*. Since the description data member was a string data type, the size of the radio unit (the number of bytes it consisted of) could easily be altered by adjusting the number of characters in it. By making the description data member very long, and the dynamic array contain a lot of records, it was possible to simulate scenarios where data sets of various sizes were being shared.

The dynamic array of radios was encoded and sent to other federates that subscribed to the array attribute instance. The subscribing federates would then store local copies of the attribute instance in their memory. If for example the federation at a specific point would consist of five federates and everybody subscribed to each and everyone's array attribute instance, each federate would then store its own attribute instance plus four other that corresponded to the other federates' attribute instances.

The sharing was done in two different ways. The first way was by using the *HLAvariableArray*, which is the standard way of sharing dynamic arrays in HLA, and the other was by using our own data type *SparseVariableArray*. The reasons for sharing the data through two different encoders/decoders were to be able to confirm that they were producing the same

result, to compare the size of data transmitted and to compare the time it took to transmit it. The *SparseVariableArray* encoder/decoder was included in our testing application *MiniTalk*.

4.4 Verification of the SparseVariableArray

When testing the *SparseVariableArray*, the same array data was sent through both the *SparseVariableArray* encoder and the *HLAvariableArray* encoder. When both updates had arrived and been decoded at the receiver side the two outcomes were compared. Then it could be confirmed that the number of elements in the resulting arrays were the same, and that every element in both arrays were the same.

In one test one element was modified and in another, all elements. In a third adding and in a fourth removing elements was tested. More test were carried out to confirm that algorithm worked as it should, which could be verified in all of the tests. It was also confirmed that no exceptions were thrown. These tests ensured that *SparseVariableArray* encoded and decoded correctly in later tests without adding the time to verify to the measured transmission time.

4.5 The Test Parameters

When evaluating the performance difference between *SparseVariableArray* and *HLAvariableArray* there were different cases to be considered, depending on:

- Array count: The total number of elements in the actual array
- Element size: The size of each element in bytes when encoded
- Elements/update: The number of elements, compared to the total, that is changed or added in the actual array, that need to be transmitted through an update

Our hypothesis was that the performance difference between the two array types likely would differ, depending on the values of these parameters. Therefore, different test cases were defined where different values of these parameters were tested.

Each parameter was tuned with values inspired by *Pitch Talk*™. For *Array count* values of 30, 250 and 1 000 were tested, and for the *Element size* each element's size kept within an array were constant, with the values of 48 and 228 bytes. For the third parameter, *Elements/update*, the extreme case when only one element had been modified and the other extreme when all elements had was tested in attempts to capture both the worst and the best cases. Cases when different fractions of elements had been changed were also tested, for example 1/3 of all the elements in the array.

4.6 The Metrics

The raw data produced in the tests were in the shape of transmission time and bytes sent per update. Out of these, more metrics were produced. These were average, minimum and maximum transmission time, the *SparseVariableArray* time presented as a percentage of the *HLAvariableArray* time, compression ratio and transmission load.

The most important metrics of this thesis were compression ratio, transmission load and transmission time. The compression ratio was calculated during the transmission of two identical updates where one was sent through the *SparseVariableArray* and the other was sent through the *HLAvariableArray*. It was extracted by comparing the total size of all packets sent through the *SparseVariableArray* to the total size of all packets sent through the *HLAvariableArray*.

The phases included in the transmission time was modification, encoding and transmission of elements at the sending federate and receiving, decoding and fetching of elements at

the receiving federate. The compression time, or the time it took handling the dictionary of indices for the *SparseVariableArray*, was not measured individually in the tests of this thesis.

Transmission load was calculated as the transmission size in bytes divided by transmission time in seconds. It is a measurement of the amount of data the encoders process per second in an update, and is used to evaluate the load on the network. A lower value of the transmission load is desirable, as it means less bandwidth on the network is necessary for the update. Transmission load was measured for the two encoding types separately.

4.7 Test Cases

The test cases consisted of one federate acting as a *sender*, one or more federates acting as *receivers* and one RTI that all federates are connected to. An overview of this is shown in Figure 4.2. The role of the sender was to update its array attribute that was used in the tests and to share the updated versions to the subscribing federates.

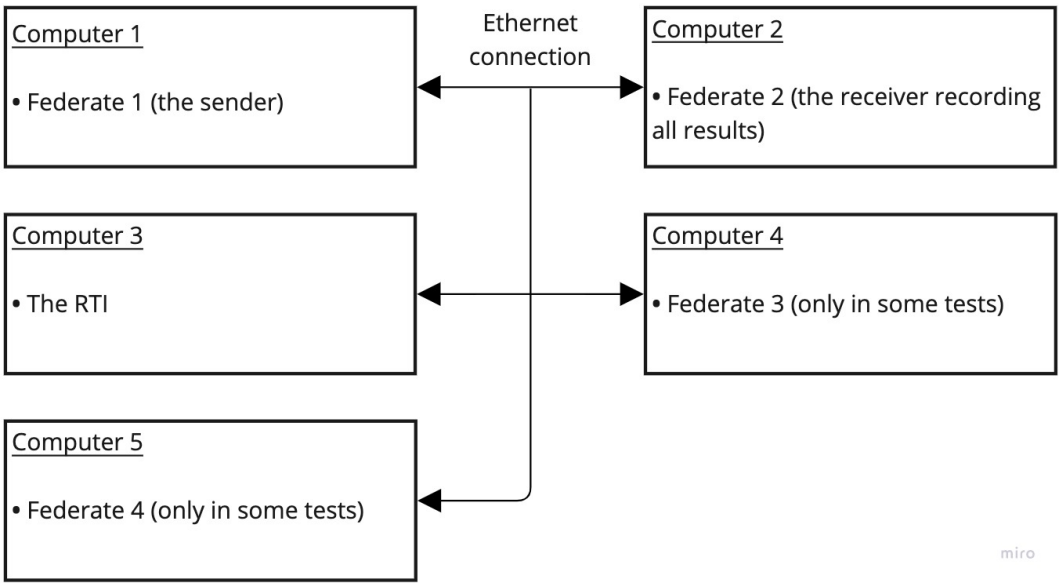

Figure 4.2: An overview of what was running on which computer throughout the tests.

All tests were carried out by using both the *SparseVariableArray* and *HLAvariableArray* individually. Even if the *SparseVariableArray* and *HLAvariableArray* were two different modules, the array attributes in the API they corresponded to would in the FOM be associated with the same type, which had the same name as the *HLAvariableArray*.

For every set of values for the parameters presented in section 4.5, ten executions were carried out. Out of the resulting transmission times the average transmission time was calculated. In each of these ten executions, 1 000 of the exact same update was sent through a `for`-loop sent from the sending federate.

In each of the 1 000 updates, the following things were done at the sender's side in the following order:

- An element was modified locally at the federate API and internally in the encoder by calling its member function `set()`. This was done for as many elements as the number the elements/update parameter was set to.
- The given encoder encoded its content into a byte-array.
- The byte-array was transmitted through the `updateAttributeValues()` method of the federate API.

Then at the receiver side, or one of them if there were more than one subscribing federate connected, all the results were recorded. To capture the transmission time of an execution containing 1 000 updates, `std::chrono::high_resolution_clock` from the standard library was used. Once the RTI detected the first update from the sender it would call `reflectAttributeValues()` method at every subscribing receiver. The following then occurred at the receiver that recorded all the results in the following order:

- A timer was started as the first of the 1 000 updates was received.
- For each receiving of the 1 000 updates, the decoder's `decode()` method was called.
- For each receiving of the 1 000 updates, each element, not only the newly received ones, of the array was copied from the decoder into the local dynamic array by calling the decoders member function `get()`.
- The last of the 1 000 updates was received and the timer was stopped.

The process was the same for both encoding and decoding techniques except the choice of encoder/decoder (*SparseVariableArray* or *HLAvariableArray*). The data copying of all elements rather than just the recently changed ones that occurred in the decode phase was done to imitate an actual use case of the *HLAvariableArray*, rather than to make the phase as fast as possible.

The reason for sending the same update as many as 1 000 times was to able to measure the transmission time more accurately. To be able to start and stop the timer at the same federate it would not have been possible to capture the time of the first update since something was needed to signal that the sending federate had started executing the test. Therefore 1 000 transmissions of the same update were made in order to make the loss of the first update negligible.

4.8 Applications Used

Except for a programming environment, OMT and RTI software were needed to perform the tests of this thesis. More specifically the applications used were *Pitch pRTI*™ and *Pitch Visual OMT*™ developed by Pitch Technologies.

Pitch pRTI™ can be used to setup the federation and to allow for the exchange of data between different federates that joins the federation execution. The tool *Pitch Visual OMT*™ has a graphical user interface and can be used for developing and adapting FOMs. These applications simplifies the controlling and the analyzing of the data being sent between federates and provides a better idea of problems and challenges that may occur when implementing a sparse encoder/decoder in a system that implements HLA.

Pitch pRTI™ and *Pitch Visual OMT*™ support the HLA IEEE 1516–2000 and 1516–2010 (also known as "HLA Evolved") standards and HLA 1.3 [14, 15].

4.9 Testing Hardware

All tests were carried out on Lenovo ThinkPad laptops. They had an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz processor which was x64-based and both had 16 GB of RAM. One computer was running a 64-bit version of Windows 11 Pro and the others were running a 64-bit version of Windows 10 Pro. The computers were connected though Ethernet cables and all communication took place through them. The link speed of the network was 1 000/1 000 Mbps.

The sending federate and the RTI were during all tests running on the Windows 11 computer and the receiving federate that recorded the results was during all tests running on the Windows 10 computer.

5 Results

This chapter begins with introducing results of the *HLAvariableArray* in section 5.1 followed by results of the *SparseVariableArray* in section 5.2. In section 5.3 results from both encoders/decoders are presented and in section 5.4 the calculated transmission load in each test case is displayed. In the chapter's last section, section 5.5, results from the test cases when four federates were connected to MiniTalk rather than two are presented.

The metrics presented in this chapter are transmission time (average, minimum and maximum), bytes sent per update, the *SparseVariableArray* time presented as a percentage of the *HLAvariableArray* time, compression ratio and transmission load.

5.1 HLAvariableArray Results

In Table 5.1, average, minimum and maximum transmission time for the *HLAvariableArray* in all test cases when two federates were connected is presented along with the number of bytes sent per update. Different numbers of elements changed per update were tested for the same array count and element size but for large arrays with the array count of 1 000, only two variations were tested due to the extra long time needed to perform those tests.

In the table it is shown that with the same array count and element size, the number of bytes transmitted per update is always the same for these results, no matter how many of the elements that have been changed for the *HLAvariableArray*. That is because the *HLAvariableArray* always send all its elements. The average transmission time is almost also the same in most cases when the array count and element size are the same. One exception is however when the array count was 30 elements and element size was 48 bytes since in this case it was faster to transmit 30 elements than to transmit 1 or 10.

There is a pattern in the results between the number of bytes sent and the transmission times when the array count is 250 or smaller. When more bytes are sent it takes more time to transmit them. On the other hand, it takes about twice as much time for the test with a medium-sized array with medium element size compared to the test with a large array with small element size, even though fewer bytes are transmitted (48 008 bytes compared to 58 004 bytes).

The time difference between the longest measured time compared to the shortest is also worth pointing out. In most cases it is below 1 second but for some of the longer test cases the difference is around 2 seconds.

Table 5.1: Result data for the *HLAvariableArray* when two federates were connected.

Description	Array count	Element size	Elements / update	Average time (s)	Min time (s)	Max time (s)	Bytes / update
Small array and small element size	30	48	1	0.390	0.245	0.404	1 448
	30	48	10	0.344	0.260	0.417	1 448
	30	48	30	0.250	0.239	0.269	1 448
Small array and medium element size	30	228	1	0.343	0.316	0.367	6 964
	30	228	10	0.330	0.300	0.410	6 964
	30	228	30	0.370	0.359	0.390	6 964
Medium-sized array and small element size	250	48	1	2.034	1.960	2.558	12 008
	250	48	50	2.320	2.201	2.409	12 008
	250	48	100	2.440	2.055	3.700	12 008
	250	48	150	1.890	1.864	1.923	12 008
	250	48	250	2.520	2.350	2.721	12 008
Medium-sized array and medium-sized elements	250	228	1	3.413	3.099	4.156	58 004
	250	228	50	3.711	3.190	4.682	58 004
	250	228	100	3.859	3.374	5.933	58 004
	250	228	150	3.230	3.112	3.326	58 004
	250	228	250	3.531	3.135	4.310	58 004
Large array and small element size	1 000	48	1	7.938	7.775	8.063	48 008
	1 000	48	1 000	7.825	7.756	7.899	48 008

5.2 SparseVariableArray Results

In Table 5.2, results captured during the test cases when two federates were connected are presented through the same metrics as in Table 5.1.

The results presented in Table 5.2 do not tell whether the encoding was sparse or dense. However, since the *SparseVariableArray* calculates which method that produces the smallest number of bytes that are to be transmitted, the number was never more than four bytes (which were used for carrying the sparse flag) larger for the *SparseVariableArray* than the number in the counterpart test case for the *HLAvariableArray*.

The number of bytes sent was highly influenced by the number of elements that had changed since the last transmission, which is a contrast to the cases of the *HLAvariableArray*. Another difference is that the number of changed elements had a relation to the transmission time. For example, in the case of the medium-sized array and medium-sized elements, it is shown that the average transmission time was 2.902 seconds when only one element had changed and 4.753 seconds when all 250 elements had changed.

In Table 5.2 the difference between the longest measured time compared to the shortest tended to be bigger in general for the *SparseVariableArray* and was almost 5 seconds for the longest case. One thing the encoders/decoders had in common though was that the transmission time got about twice as long when the array contained 1 000 elements compared to 250. Another thing they had in common was when the array count was 30 elements and element size was 48 bytes, it was faster to transmit 30 elements than to transmit 1 or 10.

Table 5.2: Result data for the *SparseVariableArray* when two federates were connected.

Description	Array count	Element size	Elements / update	Average time (s)	Min time (s)	Max time (s)	Bytes / update
Small array and small element size	30	48	1	0.287	0.202	0.357	64
	30	48	10	0.353	0.308	0.414	568
	30	48	30	0.279	0.258	0.321	1 448
Small array and medium element size	30	228	1	0.215	0.200	0.229	244
	30	228	10	0.258	0.238	0.277	2 332
	30	228	30	0.373	0.333	0.460	6 964
Medium-sized array and small element size	250	48	1	1.533	1.490	1.623	64
	250	48	50	2.268	1.583	3.301	2 808
	250	48	100	2.000	1.900	2.400	5 608
	250	48	150	2.629	1.849	2.906	8 408
	250	48	250	2.450	2.100	2.900	12 008
Medium-sized array and medium-sized elements	250	228	1	2.902	1.570	3.610	244
	250	228	50	3.367	1.805	5.369	11 612
	250	228	100	4.008	2.059	4.543	23 212
	250	228	150	4.498	2.655	5.253	34 812
	250	228	250	4.753	2.845	6.239	58 004
Large array and small element size	1 000	48	1	6.347	5.931	8.530	64
	1 000	48	1 000	9.083	8.150	12.980	48 008

5.3 HLAvariableArray and SparseVariableArray Comparison With Two Federates Connected

In this section, the average transmission times of the *HLAvariableArray* and *SparseVariableArray* are presented side by side to be able to compare them more easily. The average times of the relation between the two is also presented in Table 5.3 as the *SparseVariableArray*'s average transmission time divided by the *HLAvariableArray*'s average transmission time. For the green cells where the percentage is lower than 100, the *SparseVariableArray* produced the shortest transmission time in the corresponding test case and for the red cells where the percentage was higher, the *HLAvariableArray* produced the shortest.

The compression ratio is also presented in the table and was calculated by taking the bytes sent per update through the *HLAvariableArray*, divided the bytes sent per update through the *SparseVariableArray*. The smaller the proportion of the elements in the array that had been modified, the larger the compression ration was. Another thing to point out is that when the ratio was calculated to be 2.99 or higher in the test cases, the measured intervals were always shorter for the *SparseVariableArray* than they were for the *SparseVariableArray*. It was not however always the other way around when the ratio was calculated to be lower than 2.99, as shown in Table 5.3.

The results are also presented in Figures 5.1-5.6 where each bar chart captures a specific combination of the parameters array count and elements/update. In these cases the black bars indicate the average transmission time for the *SparseVariableArray* and the red indicate the average transmission time for the *HLAvariableArray*.

Table 5.3: Comparison of result data for the *HLAvariableArray* and *SparseVariableArray* when two federates were connected. A green cell indicates that it was faster to transmit through the *SparseVariableArray* and a red cell that it was faster to transmit through the *HLAvariableArray*.

Description	Array count	Element size	Elements / update	Sparse average time (s)	HLA average time (s)	Sparse % of HLA time	Compression ratio
Small array and small element size	30	48	1	0.287	0.390	73.63	22.63
	30	48	10	0.353	0.344	102.55	2.55
	30	48	30	0.279	0.250	111.69	1.00
Small array and medium element size	30	228	1	0.215	0.343	58.25	28.54
	30	228	10	0.258	0.330	78.26	2.99
	30	228	30	0.373	0.370	100.75	1.00
Medium-sized array and small element size	250	48	1	1.533	2.034	75.38	187.63
	250	48	50	2.268	2.320	97.75	4.28
	250	48	100	2.000	2.440	81.97	2.14
	250	48	150	2.629	1.890	139.15	1.43
	250	48	250	2.450	2.520	97.22	1.00
Medium-sized array and medium-sized elements	250	228	1	2.902	3.413	85.03	237.72
	250	228	50	3.367	3.711	90.73	5.00
	250	228	100	4.008	3.859	103.86	2.50
	250	228	150	4.498	3.230	139.24	1.67
	250	228	250	4.753	3.531	134.59	1.00
Large array and small element size	1 000	48	1	6.347	7.938	79.96	750.13
	1 000	48	1 000	9.083	7.825	116.07	1.00

In these figures it is shown that in all cases, the *SparseVariableArray* transmitted faster when only one element had changed, and in most of them, slower when all elements had. For the cases in between these extremes the results varied more. In Figure 5.1, when the array count was 30 elements and 10 elements had been updated, the *SparseVariableArray* performed slightly better than the *HLAvariableArray* when the element size was 48 bytes, but when it was 228, as shown in Figure 5.2, it was the other way around. In Figure 5.3, when the array count was 250 elements, the encoder that performed best was the same in all cases when the element size was 48 bytes compared to Figure 5.4 when element size was 228. The only exception was when 100 elements had been updated. In that case the *SparseVariableArray* performed better when element size was 48, and the *HLAvariableArray* when it was 228.

The extra transmission time overhead that the encoder came with, when all elements were transmitted can be seen from the time difference in Table 5.3 when the element size has the maximum value. The longest transmission time overhead occurred when the array count was 1 000 elements and can be calculated to 1.258 seconds. When the array count was 228 elements and the element size was 48 bytes however, the *SparseVariableArray* was faster.

When the array count was 250 elements, the *SparseVariableArray* had a transmission time that was around 139% of *HLAvariableArray*'s time. This was the case when the *SparseVariableArray* performed at its worse in terms of transmission time.

In general the *SparseVariableArray* performed better when less than half of the array's element had been updated, the element size was small and only two subscribing federates were connected.

5.4 Transmission Load

In Table 5.4, the transmission load (transmission size / transmission time) has been calculated for both the *SparseVariableArray* and the *HLAvariableArray*. Since *HLAvariableArray* always sends all elements, the transmission load does not differ as much in proportion as the transmission load of the *SparseVariableArray* when testing different numbers of elements per update. The expected transmission load for *HLAvariableArray* should be constant but due to variation in transmission time there are some variations. A notable thing is the difference in size between the encoders' transmission load. In one of the most extreme cases the transmission load of the *HLAvariableArray* was 600 times bigger than the transmission load of the *SparseVariableArray*.

The transmission load was higher when the element size was 228 bytes compared to 48 bytes. Moreover, the more of the arrays' element that had been updated in the *SparseVariableArray* case, the bigger the transmission load got. In general, the bigger the update was the larger transmission load was achieved.

Table 5.4: Comparison of the transmission load between the *HLAvariableArray* and *SparseVariableArray* when two federates were connected.

Description	Array count	Element size	Elements / update	Transmission load (sparse)	Transmission load (HLA)
Small array and small element size	30	48	1	223	3 713
	30	48	10	1 610	4 208
	30	48	30	5 188	5 794
Small array and medium element size	30	228	1	1 136	20 315
	30	228	10	9 030	21 103
	30	228	30	18 681	18 820
Medium-sized array and small element size	250	48	1	42	5 718
	250	48	50	1 238	5 176
	250	48	100	2 804	4 335
	250	48	150	3 198	3 127
	250	48	250	4 901	2 103
Medium-sized array and medium-sized elements	250	228	1	84	5 903
	250	228	50	3 449	5 176
	250	228	100	5 791	4 921
	250	228	150	7 740	6 355
	250	228	250	12 204	4 765
Large array and small element size	1 000	48	1	10	6 048
	1 000	48	1 000	5 286	6 135

5.5 Connecting More Federates

To examine how the communication changes in an HLA-based system when more than two subscribing federates connects to the federation execution, some of the tests were also carried out with four subscribing federates. The results of these tests are presented in Table 5.5 and Table 5.6 along with the previous presented results that were produced in the same way with the difference that only two federates were connected. In Table 5.5, the measured intervals are presented along with the number of bytes sent per update and in Table 5.6, the time it took transmitting through the *SparseVariableArray* is displayed as a proportion of the time it took transmitting through the *HLAvariableArray*. The results produced with four federates are also displayed in Figure 5.6.

In Table 5.6 it is easier to see which encoder that produced the shortest transmission time in each test case. If the proportion is bellow 100%, the *SparseVariableArray* produced the shortest and the cell is presented as green. If the percentage is higher than 100, it is red and the *HLAvariableArray* produced the shortest.

Figure 5.3, Figure 5.4 and Figure 5.6 shows that the transmission time for the *SparseVariableArray* grew more as the number of elements per update increased than the transmission time for the *HLAvariableArray* did. The difference between the two encoders/decoders got bigger with four federates connected and is easier to spot in Figure 5.6.

Overall the *SparseVariableArray* transmission time proportion was smaller when four federates were connected. In Table 5.6 a difference of 50 percentage points is displays. Even though the *SparseVariableArray* performed better in relation to the *HLAvariableArray* in more

cases when more federates were connected, the *HLAvariableArray* produced shorter transmission time when around more than half of the array's elements had been updated.

Table 5.5: Time differences between when two federates were connected compared to when four were.

			2 federates		4 federates		
Array count	Element size	Elements / update	Sparse average time (s)	HLA average time (s)	Sparse average time (s)	HLA average time (s)	Bytes / update (sparse)
250	228	1	2.902	3.413	1.501	4.511	244
250	228	50	3.367	3.711	2.879	3.031	11 612
250	228	100	4.008	3.859	4.030	4.744	23 212
250	228	150	4.498	3.230	4.912	4.347	34 812
250	228	250	4.753	3.531	7.258	6.688	58 004

Table 5.6: The time relation between sending through *SparseVariableArray* and *HLAvariableArray* when two federates were connected compared to the same relation when four federates were connected.

			2 federates	4 federates
Sparse % of HLA time	Sparse % of HLA time	Array count	Element size	Elements / update
250	228	1	85.03	33.28
250	228	50	90.73	91.69
250	228	100	103.86	84.95
250	228	150	139.24	113.00
250	228	250	134.59	108.54

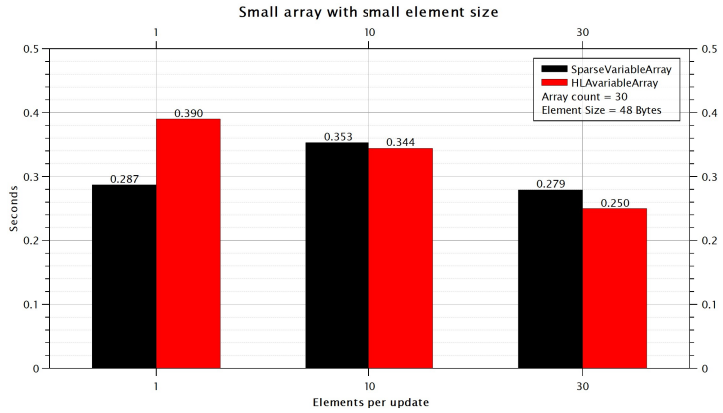

Figure 5.1: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 30, each element is 48 bytes large. Tested with 1, 10 and 30 updated elements per update.

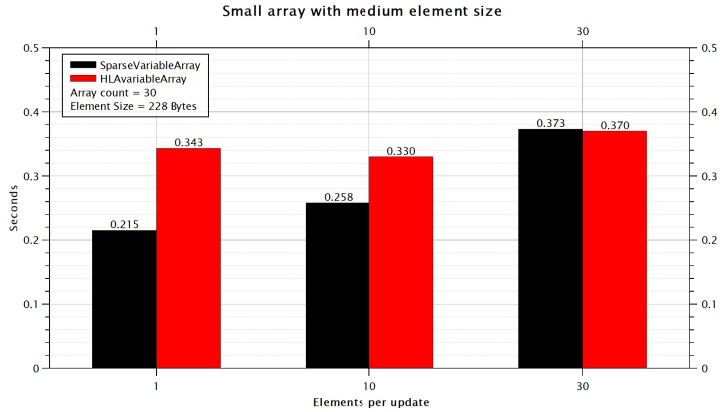

Figure 5.2: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 30, each element is 228 bytes large. Tested with 1, 10 and 30 updated elements per update.

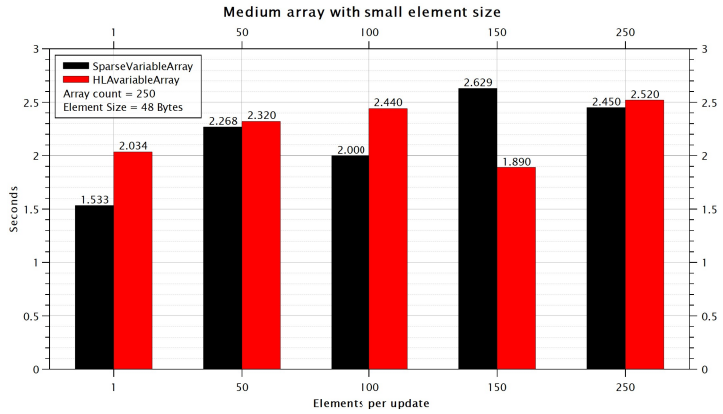

Figure 5.3: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 250, each element is 48 bytes large. Tested with 1, 50, 100, 150 and 250 updated elements per update.

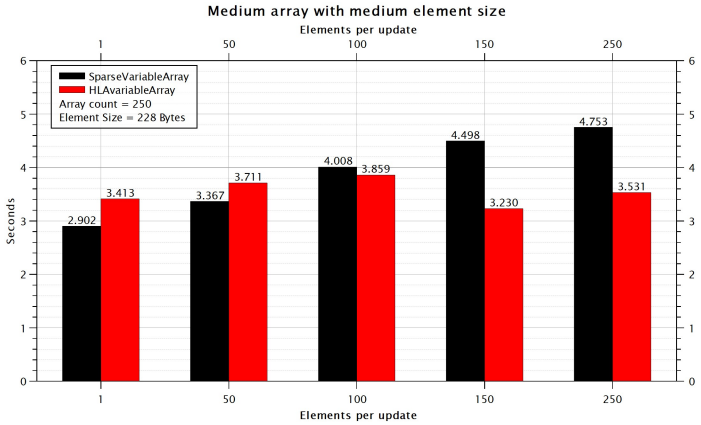

Figure 5.4: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 250, each element is 288 bytes large. Tested with 1, 50, 100, 150 and 250 updated elements per update.

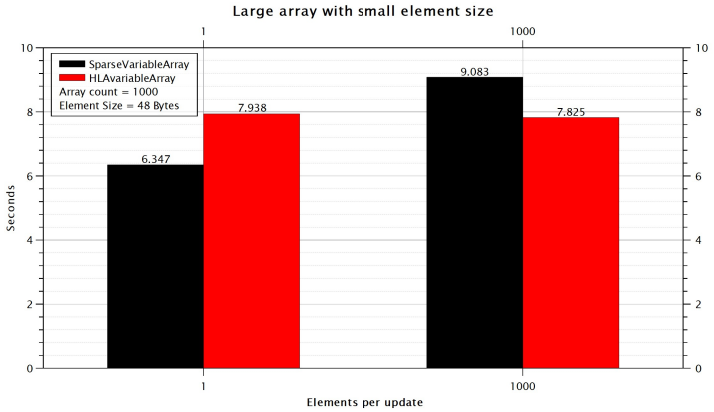

Figure 5.5: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 1 000, each element is 48 bytes large. Tested with 1 and 1 000 updated elements per update.

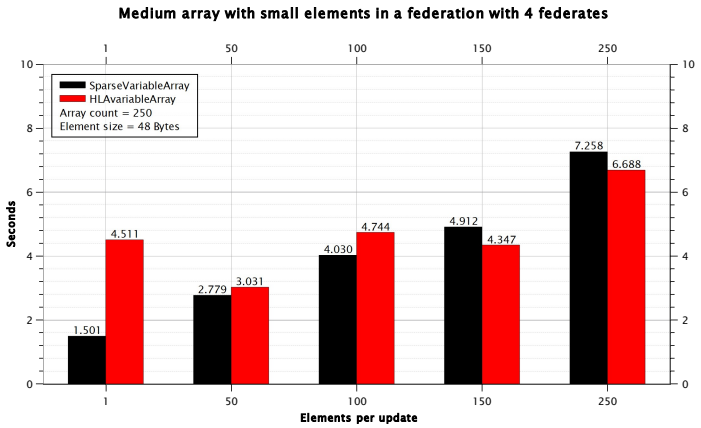

Figure 5.6: Average time of 10 tests of sending and receiving 1 000 updates with *SparseVariableArray* and *HLAvariableArray*. With an array element count of 250, each element is 48 bytes large. Tested with 1 and 1 000 updated elements per update, in a federation with four different federates, each run on four different computers.

6 Discussion

In this chapter, the results, their meaning and their connection to the research questions are discussed followed by thoughts about the choice of method for this thesis.

6.1 Research Question 1

In the first subsection, array properties that for which the proposed TSE technique may contribute to shorter transmission time is discussed, and in the second, the ones that may contribute to smaller transmission size are discussed.

6.1.1 Transmission Time

Something that seems to be recurrent in the results is that the smaller the proportion of the arrays element that have been modified is, the more transmission time is saved with the *SparseVariableArray*. For the federation executions in which only two federates were connected, this saved transmission time was however quite short and in some cases maybe even insignificant.

An advantage for the *HLAvariableArray* is that it only needs to calculate the expected encoded length once, while the *SparseVariableArray* needs to do it twice, once for getting the expected encoded length if it was to encode densely (like the *HLAvariableArray*) and once for if it was to encode sparsely. Since this is done every time the *SparseVariableArray* encodes something, this procedure will always take more time for the *SparseVariableArray* but the amount of time is dependent on the number of elements changed. This would suggest that the number of elements is a property that contributes to shorter transmission time as an answer to RQ1.

For both the *HLAvariableArray* and the *SparseVariableArray* the transmission time doubled when the array count was set to 1 000 instead of 250 even though significantly less bytes per update were transmitted. Since the *SparseVariableArray* only transmitted slightly faster than the *HLAvariableArray* when updating 1 out of 1 000 elements it was probably not the time spent on the Ethernet network that consumed the most time. It would be more likely that the most time was spent copying the data from the decoder to the local array at the application layer of subscribing federate. When removing the code executing all copy operations at the

subscribing federate the measured transmission time was around 10% to 30% of the time measured with that code for the test case mentioned above.

Because of this it seems reasonable to conclude that the bottleneck when few federates are connected probably is the copying, or the fetching of elements. Thereby the size of the array would have a large and significant impact on the transmission time as an answer to RQ1.

6.1.2 Transmission Size

Similarities between the transmission time and the compression ratio can be observed in Table 5.3. When two federates are connected it seems like transmissions through the *SparseVariableArray* often tend to be faster when the compression ratio is 2.99 or higher as an additional answer to RQ1. In other words, when less than 1/3 of the arrays elements have been updated. When more than 1/3 of them have been updated, the *SparseVariableArray* does not always seem to be faster. For this to be more reliable however, it is likely that more test cases would be needed, especially given the few amount of test cases with an array containing 1 000 elements. Moreover, regardless whether the data is sent through the *HLAvariableArray* or the *SparseVariableArray* all the elements of the array needs to be copied from the decoder, so that should not have much impact.

When sending data over TCP it is usually better to send fewer larger packets than several smaller ones [16]. One reason for that is that each packet demand a certain number of bytes as overhead, so the more packets needed, the larger the overhead. In Table 5.2, when the array count was 30 elements and only one element was updated, it went faster for the *SparseVariableArray* to send updates with 228 bytes compared to 64, which might seem a bit strange. An explanation could be temporal network interference and operating system tasks, which in this case may had been able to make a difference for the transmission time, since in both cases probably only one packet was sent. When the data was bigger than 1 448 bytes however, it was probably split up into several packets.

6.2 Research Question 2

In this section it is discussed how the transmission time and transmission size differ when using the proposed TSE technique compared to when using the standard technique described in HLA. First the time differences are discussed, then the compression ratio followed by a discussion of differences when more federates are connected.

6.2.1 Time Differences

In the cases when the array count was smaller and the elements size was bigger, it is easier to examine the gain of smaller transmissions over the network even if a large proportion of the time still is spent on copying elements at the decode stage. For example, in Table 5.3, with a 250 element large array and the element size 228 bytes, the transmission time for the *SparseVariableArray* was about 85% of the *HLAvariableArray* even when only one element was transmitted. So in federations where only two federates are connected the data copying stands for a great proportion of the total transmission time.

The *SparseVariableArray* encoder always uses some extra time to calculate the length of the corresponding methods of encoding. That is probably why the *HLAvariableArray* almost always performed shorter transmission time when most elements were to be sent because the time it took to handle the indices in those cases turned out to be longer than the time that would have been needed for encoding the remaining unmodified elements. For RQ2, this means that the transmission time tends to be a little shorter for the *HLAvariableArray* when near 100% of the array is updated while transmission size will be exactly the same or smaller for the *SparseVariableArray*.

The test cases where almost all elements were sent and the encoder encoded sparsely seem to have been the worst cases for the *SparseVariableArray* A possible improvement for the *SparseVariableArray* could be to send all elements, when a certain proportion of the elements has changed, perhaps close to 2/3 of the array's elements rather than all of them. In that way the encoded length calculations could be avoided in those cases.

Sometimes the shortest measured interval for a specific test case differed a lot compared to the test case's longest. The longer the test case was the bigger the difference could be. This is not strange since there were a couple of things in our tests that was hard to have complete control of, like background activities by the operating systems and interference on the network. The impact of these factors were reduced by doing the same test 10 times and then calculate the average transmission time.

Running the tests 10 times in a row did not seem to eliminate the problem with unknown background activities and interference completely though. For the test case when the array count was 30 elements and element size was 48 bytes, the reason for that it was faster to transmit 30 elements than to transmit 1 or 10 for both encoders could have been because of CPU consuming activities executing in the background when the test cases were carried out. Why the transmission time was shorter for both encoders was probably because every test case first was run with the *SparseVariableArray* and immediately after run with the *HLAvariableArray*.

6.2.2 Compression Ratio and Space

For the compression ratio, it was always one for the *HLAvariableArray* since all elements always are transmitted. For the *SparseVariableArray*, it grew as the number of elements changed per update decreased. There will be extra indices occupying space corresponding to each element sent in the transmission up to the point where the *SparseVariableArray* finds it more efficient, in terms of space, to transmit densely. Thus, the answer to RQ2 regarding the compression ratio is that the smaller the proportion of the elements that are updated is, the more the two encoders/decoders will differ in transmission size. The larger the proportion is, the more similar the size of the transmission will be for the encoders/decoders.

There is also extra space needed on the computer that executes the encoding procedures. Mainly the extra space corresponds to the search tree of indices referring to all modified elements in the *SparseVariableArray*. Since the elements of the search tree that was implemented throughout this thesis consisted of a 64-bit integer representing the index and a few additional pointers and bits, the overhead size would probably in most cases be insignificant. If the array was large enough however, a notably amount of memory would probably be needed.

6.2.3 Four Connected Federates

Additional answers to RQ2 can be found when examining the test cases when four federates were connected instead of two. In cases where the *HLAvariableArray* had longer transmission time compared to when two federates were connected, the difference in time between the encoders/decoders increased and in cases where the *SparseVariableArray* had longer, it decreased or changed in a way so that the *SparseVariableArray* became faster than the *HLAvariableArray*. Thus it seems like the more federates that connects, the faster the *SparseVariableArray* transmits relative to the *HLAvariableArray*.

Why the transmission time for the *SparseVariableArray* was growing more as the number of elements per update increased than the time for the *HLAvariableArray* did could be because of the network bandwidth. With four federates more traffic occurs on the network while the time it takes copying elements in the decoding phase remains the same. Of course, with more federates decoding, more copy operations will occur, but since they are executed on different computers, it is done in parallel.

6.3 Research Question 3

In this section it is discussed how the transmission load differ when using the proposed TSE technique compared to when using the standard technique described in HLA.

The difference between the transmission load the both encoders/decoders produced, which RQ3 seeks answers for, was in all test cases significant and transmitting through the *HLAvariableArray* always produced larger transmission load. We interpret the transmission load difference between the *HLAvariableArray* and the *SparseVariableArray* as an indicator of that the bottleneck was not the network bandwidth when only two federates were connected. That is because even if the amount of transmitted data was reduced with a certain amount through the *SparseVariableArray* compared to the amount that the *HLAvariableArray* would have produced, the transmission time was not close to change as much as it would have if the transmission on the network would have been the only time consuming procedure.

The transmission load produced in our tests, displayed in Table 5.4, was greater when the element size was bigger (228 bytes instead of 48) for both encoders, or in other words, when the size of each update was larger. This probably has to do with that we get larger and fewer packets when we transmit the data.

6.4 Method

Since the *SparseVariableArray* almost contains the same set of member functions as the *HLAvariableArray* (like for example the `set()`, `get()`, `addElement()`, `encode()` and `decode()` functions), it is likely that replacing the *HLAvariableArray* with the *SparseVariableArray* would not imply a lot of code editing since they are very similar in the ways they can be used. One exception would be the pointer functionality which the *SparseVariableArray* does not have because of the delimitations of this thesis.

As explained in subsection 4.2.2, *SparseVariableArray* decides to encode sparsely or densely based on the number of bytes each method produces. Thus, it is guaranteed that with TSE, the transmission size is either smaller, or at most 4 bytes larger, than the transmission size of the original encoding. Less data transferred can also help reduce the transmission time.

A downside to the implementation of deciding whether to encode sparsely or not, is that the encoded size must be computed each time by looping through the list of indices and list of elements. Because individual element sizes can vary (for example an array of strings) every element's encoded size must be calculated. The larger the array, the larger the time of encoding sparsely becomes, which affects the transmission time. If it is known that the element size is constant, a more efficient way would be to calculate and compare the element size and multiply it with the number of changed elements, and total elements respectively.

A more accurate way of measuring the strains on the network would be to calculate the transmission load with transmission time of only the transferring of data on the network. The problem would be that the transfer time per a single update would be so small that the measurement errors would be so big that they would lead to inaccurate measurements when having to measure it 1 000 times and calculating the sum for every test. The less time spent on extracting and copying of elements, the more accurate transmission load can estimate the bandwidth the encoders occupy.

6.4.1 Tests and Measurements

If the pointer based functions had been implemented and used, the elements would have been updated directly, and extracting the elements would not have been necessary. Because it seems that extracting the elements is an expensive operation, it would be desirable to make tests and measurement of a pointer implemented version of *SparseVariableArray*, as it could have prevented the high fluctuation in transmission time. Thus producing more consistent results and comparisons. This would also make transmission load a better estimation of

the load on the network. If both *SparseVariableArray* and *HLAvariableArray* would have used pointers, the answer to RQ2 would probably be different. Because both encoders in the tests extracts element identically (and should in theory spend the same amount of time extracting and copying elements), only the overall transmission time would decrease, and not necessarily the difference between the encoders. Thus the value of the transmission time would maybe not differ more between the encoders, but with an overall shorter time, the time proportion between the both encoders/decoders might had been more significant.

However, *HLAvariableArray* does support both storing and owning elements, and pointers. Pointers also implies more responsibility for the user/API as they require very careful memory management and while it automatizes changing elements, adding elements still require interaction with the encoder. Hence, it is not only a matter of performance, but also convenience and ease of use.

In each of the test cases throughput this thesis, 1 000 updates were always transmitted, but there are a lot of other ways in which the tests could have been carried out. For example, sending more or less bytes per update could have been tested or examining how many updates that would have been transmitted and handled within a certain time interval could have been done. The transmission load could also have been produced by adjusting the system to always send for example 10 updates every second. Then there would probably have been a clear relation between the update size and the time aspect.

The relative large fluctuation in transmission time could also have been investigated further. Had the extraction of elements been separated throughout the tests it would have been possible to investigate whether the fluctuation depended on the network connection, on the extraction of elements (which is more related to the CPU) or both.

Because transmission load is dependent on the transmission time, it is important that the transmission time is measured only as encoding and transmitting the packets for both of *SparseVariableArray* and *HLAvariableArray*. For example, if there are other time consuming procedures during the transmission time for one of the encoders, the results would become invalid. There is also a scenario in which the transmission load would not provide honest results. That would be when all of the available network bandwidth is used because then both the *HLAvariableArray* and the *SparseVariableArray* would produce the same, or close to the same, transmission load. That is because the bottleneck in in these cases would be the bandwidth.

6.4.2 Test Parameters Used

The values the parameters (see section 4.5) *array count* and *element size* had in the tests were chosen because they represented likely values from scenarios from the application *Pitch Talk*™. The varying of these parameters allowed for a selection of arrays with different properties relevant for answering RQ1.

Because TSE was also designed to work with HLA in general, more tests with larger arrays and element sizes would have been good to get more versatile results. The parameter values that were prioritized throughout this thesis were similar to the ones relevant for Pitch Technologies but in future work, other parameter values should also be tested to get a broader understanding of the effects of TSE.

As seen in the results, the transmission times of small arrays with small or medium elements, are very small. For these tests running more than 1 000 transmissions would have increased the total transmission time for both *SparseVariableArray* and *HLAvariableArray*. This might have been desirable, since longer transmission times might have showed the differences between the times better. The same amount of transmissions per test were used in order to make all test cases more comparable to each other.

6.4.3 Scalability

In this thesis, tests with a maximum of four federates in a federation were tested. However, the method itself is not limited to four federates, and to further investigate RQ2 more federates can be tested. Testing with more federates would also allow for investigating TSE when potentially neither the network nor the computer hardware (e.g., CPU and main memory) are the bottleneck, and how it affects the answer RQ2.

Multiple federates can encode through the TSE technique since *SparseVariableArray* and federates are independent from each other. Nor were any issues with scalability, in terms of number of federates and the size of the array, observed during the testing. A drawback for *SparseVariableArray* is that each attribute instance must have its own instance of its encoder respectively its decoder. This may not be optimal in a way however because for every dynamic array attribute, two instances of the encoder class must exist, one for encoding and one for decoding.

For the tests in this thesis, each execution of *MiniTalk* (and thereby each federate) was executed on a separate computer in order to mitigate the application's dependency on the computer hardware. However, it might not be a viable option when testing with tens or hundreds of federates. Instead, if necessary, running multiple instances of *MiniTalk* on the same computer is possible to achieve an even higher numbers of federates. At the point in time when this thesis was carried out, a limit to *MiniTalk* was that no more than one federate could run from the same instance of the application, thus the number of federates was limited to the number of instances of *MiniTalk*.

The largest array that occurred in the tests of this thesis contained 1 000 elements. The concept of TSE itself is independent on the number of elements and the size of the elements. Even though the *SparseVariableArray* encoder is limited to a maximum elements of a 32-bit unsigned integer, there are probably more problems with hardware/network resources than there are with exceeding the maximum number of elements within a single array. A 32-bit integer is in compliance with the HLA standard, and a larger array is therefore not of interest. There should not be any problems with scalability in terms of array size, as far as the encoding method is concerned, nor any problems with arrays larger than those tested in this thesis. The *SparseVariableArray* is also independent on the type of element as long as it is of an HLA data type (just like *HLAvariableArray*) and should therefore work with any HLA data type that can be used in the FOM.

6.4.4 Problems With Encoding Sparsely

A problem with TSE when used for HLA is the case where sparse data is transmitted and a decoder which has not yet to receive all data, starts to decode the sparse data. Currently *SparseVariableArray* (as a consequence of the implementation of *HLAvariableArray*), constructs the HLA-type elements with default initialized values. For example, an element of an *HLAinteger64BE* would give an integer with value 0.

There was no time to implement or test this case, but the proposed solution would be to indicate to the user that sparse data was received before the complete data, and let the user request a full update. For example, in order to verify that sparse data is received before the complete data has been received, *SparseVariableArray* can check if any data, where the *sparse flag* set, is decoded before ever receiving data where the *sparse flag* is not set. Unfortunately this requires interaction from the user, making the encoder less self-contained, but it is better to mark such data invalid than to continue to use it. Therefore the user should handle this situation, since the user would most likely need to request all data, which the encoder would not be able to do by itself.

6.5 The Work in a Wider Context

As for the "product" itself, this work focuses on HLA that is used often but not exclusively, within the military sector. HLA is just a standard and can be used within any simulation of any sort. This work, or the work on the HLA standard is not considered to be unethical by itself. It is true that it is used for military, which in turn might use simulations for situations more controversial from an ethical stand point. However there are other systems of simulations, non-related to military, like space simulations. It can be argued that working with HLA is no more unethical than writing any piece of software, or technology as such. It all depends on how it is used, and certainly anything can be used unethically, but this work, software and technology does not in any way promote misuse, nor lean towards using it in any specific way.

Many areas have ethics related to production. For example, it can be argued that a lamp itself is not unethical, but maybe the production of the lamp is. In our case, we are writing a piece of software, thus we control the process of producing the "product". Of course one can look at the origin equipment used, if they were produced in an unethical manner or if they are bad for society in any way, but that is a bit far fetched in our opinion and is so general it can be applied to anything and indefinitely.

Our work specifically might not affect society much, since this work is only relevant for a few people. However simulation itself can be quite a big element in society, and better use of resources is generally positive. For example, simulations can simulate scenarios that can not be created in real life, which can lead to development of technological advancements.

7 Conclusion

In this thesis, an alternative encoder/decoder to the *HLAvariableArray* with the name *SparseVariableArray* has been proposed and tested. The TSE technique it implements is inspired by sparse encoding/decoding and reduces the data transmitted in an HLA-based system. A series of tests were carried out to answer the research questions.

The findings of this thesis suggest that the *SparseVariableArray* performs better in terms of transmission time than the *HLAvariableArray* when few elements have changed and need to be transmitted as well as when the data sent during each transmission is large. They also suggest that it performs worse than the *HLAvariableArray* when a large fraction of the array's elements have changed and when the data sent during each transmission is small in size. When for example 1 of an array's 30 elements with an element size of 228 bytes was updated and sent, the *SparseVariableArray*, to answer RQ2, performed the whole transmission process in 58% of the time it took for the *HLAvariableArray* to perform it. On the other hand when the array had 250 elements with the same element size and 150 of the elements were updated and sent, the *SparseVariableArray* performed the process in 139.24% of the time it took for the *HLAvariableArray* to perform it when two simulation systems were connected to the execution.

The transmission load produced by the *HLAvariableArray* was in all test cases bigger than the one produced by the *SparseVariableArray* in answer to RQ3. The compression ratio was however always bigger or the same for the *SparseVariableArray*. When a small fraction of the arrays' elements in the tests was sent to other federates the compression ratio was many times bigger compared to when all elements were sent. Because the transmission load of the *SparseVariableArray* was smaller and that the transmission time was not much shorter as expected, a conclusion that was made was that the bottleneck of the performed tests was the time the CPU of the computers spent copying elements from the decoder at the receiver side rather than network bandwidth.

Properties of an array attribute that is being shared within an HLA-based system that contributes to shorter transmission time and smaller transmission size, as asked for in RQ1, were array size and number of modified elements. The array size was important since all elements were always copied from the decoder when an update had been received no matter which of the examined decoders that was used. The number of modified elements mattered especially when it came to compression ratio and number of bytes transmitted, but also transmission time.

Another conclusion regarding RQ2 is that the more simulation systems that are connected to the execution, the bigger the difference in transmission time gets in cases when the *SparseVariableArray* produces the least amount of data.

The spacial overhead of the proposed technique corresponded to an extra array of indices in the *SparseVariableArray* object and in the packets being transmitted. The size of that array did not have a significant impact on the overall performance however. For the CPU calculation time, the *SparseVariableArray* spent some extra time calculating and deciding if the data should be sent compressed or not which had an impact on the performance in some cases.

7.1 Future Work

There are many possible directions in which this work could be continued. For example, since a lot of time is spent copying and extracting elements, there are significant optimization possibilities regarding how elements are connected to the encoder and how data is managed once decoded. A way to avoid the copying to a greater extend could be to use pointers inside the encoder and/or decoder. This would likely imply modifying the *SparseVariableArray* and handing over more responsibilities to the API using the encoder and decoder.

In reality there are of course a lot of cases when more than four federates are involved. A suggestion for future work would thus be to test if the proposed TSE technique performs better in larger HLA-based systems with more federates connected, as the findings of this thesis suggest.

A factor that likely impacts whether it is profitable to use *SparseVariableArray* compare to the *SparseVariableArray* is the bandwidth of the network. A hypothesis would be that the lower the bandwidth is, the better the *SparseVariableArray* performs in relation to how the *HLAvariableArray* performs. Another interesting continuation to this work could be to examine the sparse array methodology when sending updates in an HLA-based system through UDP instead of TCP.

The performance impact of the functionality for inserting and removing elements anywhere in the object instance of the encoder/decoder was not tested in this thesis. There were tests that verified that the functionality produced correct results but none that tested the transmission time impact for example. Testing this functionality would thus be a suggestion for future work.

In this thesis, it was not examined whether the elements of an array could be compressed in them selves or not and what the possible effects of that would be. Multidimensional arrays could be an example of a case when that would be interesting to examine.

Bibliography

[1] J.S. Dahmann, R.M. Fujimoto, and R.M. Weatherly. "The DoD High Level Architecture: an update". In: *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*. Vol. 1. 1998, 797–804 vol.1. DOI: 10.1109/WSC.1998.745066.

[2] F. Kuhl, R. Weatherly, and J Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. 1st ed. New Jersey: Prentice Hall, 1999. ISBN: 0130225118.

[3] NMSG Exploratory Team 046. *Recommendations for the revision of IEEE Std 1516-2010 (HLA Evolved)*. NATO STO, 2022.

[4] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Federate Interface Specification". In: *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)* (2010), pp. 1–378. DOI: 10.1109/IEEESTD.2010.5954120.

[5] C. Lin, Y. Chung, and J. Liu. "Data distribution schemes of sparse arrays on distributed memory multicomputers". In: *Proceedings. International Conference on Parallel Processing Workshop*. 2002, pp. 551–558. DOI: 10.1109/ICPPW.2002.1039777.

[6] D. Yan, T. Wu, Y. Liu, and Y. Gao. "An efficient sparse-dense matrix multiplication on a multicore system". In: *2017 IEEE 17th International Conference on Communication Technology (ICCT)*. 2017, pp. 1880–1883. DOI: 10.1109/ICCT.2017.8359956.

[7] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Framework and Rules". In: *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)* (2010), pp. 1–38. DOI: 10.1109/IEEESTD.2010.5553440.

[8] Pitch Technologies. *Free HLA Tutorial and Software*. https://pitchtechnologies.com/hlatutorial/. Accessed: 2022-03-29.

[9] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Object Model Template (OMT) Specification". In: *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)* (2010), pp. 1–112. DOI: 10.1109/IEEESTD.2010.5953408.

[10] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Other Titles in Applied Mathematics. SIAM, 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003. URL: http://www-users.cs.umn.edu/%5C~%7B%7Dsaad/IterMethBook%5C_2ndEd.pdf.

[11] Chun-Yuan Lin, Yeh-Ching Chung, and Jen-Shiuh Liu. "Efficient data compression methods for multi-dimensional sparse array operations". In: *First International Symposium on Cyber Worlds, 2002. Proceedings.* 2002, pp. 62–69. DOI: 10.1109/CW.2002.1180861.

[12] James King, Thomas Gilray, Robert M. Kirby, and Matthew Might. "Dynamic Sparse-Matrix Allocation on GPUs". In: *High Performance Computing*. Ed. by Julian M. Kunkel, Pavan Balaji, and Jack Dongarra. Cham: Springer International Publishing, 2016, pp. 61–80. ISBN: 978-3-319-41321-1.

[13] Y. Lin C, S Liu J, and C. Chung Y. "Efficient representation scheme for multidimensional array operations". In: *IEEE Transactions on Computers*. Vol. 51. 3. 2002, pp. 327–345. DOI: 10.1109/12.990130.

[14] Pitch Technologies. *Pitch pRTI*. https://pitchtechnologies.com/prti/. Accessed: 2022-03-29.

[15] Pitch Technologies. *Pitch Visual OMT*. https://pitchtechnologies.com/visual-omt/. Accessed: 2022-03-29.

[16] Jari Nieminen, Jouni Karvo, Pasi Lassila, and Markus Peuhkuri. "Impact of heterogeneous packet sizes on flow fairness". In: *2009 International Conference on Telecommunications*. 2009, pp. 248–253. DOI: 10.1109/ICTEL.2009.5158653.