B. TECH. PROJECT REPORT

On

Marshalling of Large Data into Data-Exchange Formats

BY

Kalyan Garigapati

Cse140001011

DISCIPLINE OF COMPUTER SCIENCE ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

December 2017

Marshalling of Large Data into Data-Exchange Formats

A PROJECT REPORT

Submitted in partial fulfillment of the
requirements for the award of the degrees

of
BACHELOR OF TECHNOLOGY
in

COMPUTER SCIENCE ENGINEERING

Submitted by:
Kalyan Garigapati
Cse140001011

Guided by:
Dr. Somnath Dey
Assistant Professor
IIT Indore

INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2017

CANDIDATE'S DECLARATION

We hereby I declare that the project entitled **Marshalling of large data into data-exchange formats** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in Computer Science Engineering completed under the supervision of **Dr. Somnath Dey, Computer Science Engineering,** IIT Indore is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Signature and name of the student with date

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

Signature of BTP Guide with dates and their designation

Preface

This report on Marshalling of large data files into data-exchange formats is prepared under the guidance of **Dr. Somanth Dey**.

I have interned Seller Technologies [8] Team, at Amazon India, during the seventh semester. During my six month internship at amazon, I had worked on various softwares. I had chosen to present my work on the Warehouse Management System as my B. Tech Project. Apart from my internship work, I had further explored and contributed to it.

Through this report I have tried to give a detailed design of an architecture for marshalling very large files, even in limited memory constraints. In addition, I have also added necessary logical procedures to make this architecture support symmetric key encryption.

I have comprehensively explained the entire implementation in a step-by-step manner, and also added flow charts where ever necessary. Also I have collected multiple datum on every necessary events and CPU resource details to show accessible results.

Kalyan Garigapati
B.Tech. IV Year
Discipline of Computer Science Engineering
IIT Indore

Acknowledgements

I wish to thank **Dr. Somnath Dey** for his kind support and valuable guidance throughout the project duration, enlightening me with useful insights.

I would also like to thank my mentor **Sumit Goswami,** Software Development Engineer at Seller Technologies, Amazon Inc., for his remarkable contribution to this project.

It is his help and support, due to which I was able to complete the design and implementation along with the technical report.

Finally, I would like to sincerely thank everyone who knowingly or unknowingly helped me in this project.

Without their support this report would not have been possible.

Kalyan Garigapati

B.Tech. IV Year

Discipline of Computer Science Engineering

IIT Indore

Abstract

In this paper, I have presented a study of marshalling large data files containing repeated data-item elements into data exchange formats like XML, JSON and others. From many years, extended markup language (xml) has emerged as the core of the Web services architecture, and is playing crucial roles in messaging systems, databases, and document processing. However, the XML data can be very large, taking up to peta-bytes of storage. In many cases such a large xml data is made from a data model, which has one/multiple repeated elements of same type. This property of large data can be taken as an advantage and a strategy to marshal such information via streaming can be designed.

In this report, I have proposed an efficient method for parsing and creating a data transfer formatted file from a very large data (for example: a csv data with 1 billion entries) in limited memory constraints.

I have split the marshalling of large files into three parts, marshalling message headers, stream marshalling of the repeated data type and finally marshalling the message footers. Further I have taken care of memory constraints by constantly uploading the marshaled data or by saving them to a file.

This algorithm is implemented for XML marshalling with the marshalled data uploaded to Amazon's S3 storage service [9] and also implemented for JSON marshalling with the buffered marshalled data stored in a file on the hard-disk. This implementation is also put into real use by the end of its successful implementation and testing. This algorithm is now being used for syncing two csv images of Amazon .Inc's warehouses.

All the analytic data corresponding to the CPU resources were collected while the algorithm was under execution. This is also repeated for the currently available normal marshalling algorithms. Both the analytics were compared and reported as graphs. All the advantages and disadvantaged of using this algorithm were reported in the results.

Index Terms- marshaling an xml, means assembling raw data into data-transfer formats

Contents

List of Figures & Tables

Figures

Tables

List of Technical Vocabulary

Abbreviations

CSV	Comma Separated Values
XML	Extended Markup Language
JSON	JavaScript Object Notation
POJO	Plain Old Java Object
WMS	Warehouse Management System
AES	Advanced Encryption Standard
W3C	World Wide Web Consortium
SSE	Server Side Encryption
XSD [6]	XML Schema Definition
DMA	Direct Memory Access
SKU [7]	Stock Keeping Units

Definitions

Marshalling

It is the process of transforming the memory representation of an object to a data format suitable for storage

Un-marshalling

Reverse of the marshalling process, i.e. transforming the document to memory representation of an object

Disposition

In the context of this paper, disposition represents the state of an inventory item in a vendor's warehouse.

Chapter 1
Introduction

This article outlines a stepwise walkthrough for marshaling an xml document from a large raw data (example, CSV data). In addition to this, we will also look into the situations where we create a stream convertor which takes a reader to read the raw data file and output will be a stream which gives marshalled data in xml format. We will also cover the case where the output xml data has to be an encrypted file. In a typical scenario of software industry, large data from one service to other is shared by saving the data on an online storage service and just sending the address & path of the file. In few cases if the data contains critical customer data, file will be encrypted with a common cipher key.

Currently JAXB framework (Java Architecture for XML Binding) provides necessary API to marshal a POJO (Plain Old Java Object) into extended markup language. But the size of the object was constrained to memory availability. Moreover, the resultant xml will be stored in memory again. In next chapters, we will be using JAXB framework to convert partial data to xml.

Sample Input:

```
 WAREHOUSE            CORE      <quantity in dispositions> . ..
 Internal Id         Item Id

dellxps4Z2dd2,  laptopr24r3gu5 ,12,3,0,0,0,3,0,0,0,0
smsung8beAd23,  laptops5yg3efg ,10,0,0,1,0,0,0,0,0,4
rcChoppr8e3eq,  toyrca4Ur3r2gb ,45,0,0,1,0,4,0,0,2,0
woodendsk2da2,  furnitd3BGLft4 ,32,0,0,0,0,0,0,0,0,0
watrbottl7sw8,  bottleda98salL ,42,0,0,0,2,0,0,0,5,0
```

Figure 1.1: raw data which has to be marshaled to xml

In the Fig. 1.1, there are 12 columns in the inventory image. First two columns correspond to the unique Ids of an item. All the remaining columns contain the quantities of the product in different dispositions. This total tables is called inventory image.

Sample marshalled data:

```
<?xml version="1.0" encoding="UTF-8"?>
<transmission >
        <message id="1">
                <InventorySyncMessage>

                        <inventoryControlDetail inventoryControlType="FULL"/>

                        <inventoryItemData id="dellxps4Z2dd2">
                                <disposition id="damaged" quantity=21/>
                                <disposition id="expired" quantity=2/>
                                .
                                .
                                .

                        </inventoryItemData>

                </InventorySyncMessage>
        </message>
</transmission>
```

Figure 1.2: Sample xml representations of raw data shown in fig. 1

The above figure Fig. 1.2 contains a transmision in extended marklup language. This transmission corresponnds to the sending of a message from one service to other. This transmission may contain multiple messages. Each message element contains a type of message in it. In the current scenario, the message is having 'InventorySyncMessage'.

Figure 1.1 shows the sample raw data which our algorithm will have to parse for marshalling. The repeated element in this data set is ***InventoryItemData***, which can be seen in the Fig. 1.2. The Fig 1.2 displays a sample output of our algorithm, i.e., the marshalled raw data.

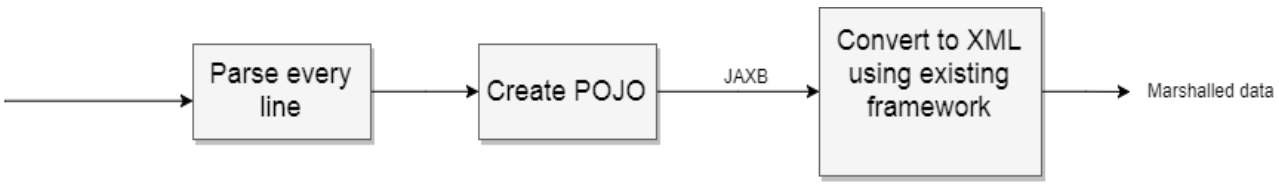


Figure 1.3: steps followed in general algorithm

The image shows the flow of control in general marshalling algorithm. First each line is parsed, followed by creation of POJO. Finally, The POJO is converted to XML.

A general algorithm trails the following process as shown in Fig. 1.3:

a) Parse each row in the raw input (.csv) file
b) Create a memory object (POJO in the current implementation) from the parsed data
c) Define rules for the conversion of memory object to data-transfer format and convert it using any external framework.

1.1 Motivation

Context: In a warehouse management platform, we call the set of availability details of all the inventory a warehouse contains at a particular instant of time as inventory image.

All the online goods selling platform has a customer's portal which enable outside buyers to place orders on available products and a warehouse management system which is mostly used by the seller, and this management system allows seller to add/delete/change availability status of any of the product in his warehouse. A typical online goods selling platform will have multiple sellers on their side and will maintain a global inventory image, and all the customer order will be based on this image of inventory, but the problem arises when there is a discrepancy between the image maintained by warehouse management on seller's side and image maintained by the core of goods selling platform.

How the discrepancy arises:

Let's suppose the seller side system's image contains product A with 1 unbound item and 2 items which are bound to previous customer orders (but still are not picked to ship) and the same image of this seller's warehouse in the core platform. When a customer orders product A, the core will look at the image it had and finds it had an item in unbound state, so it places the order and sends order successful notification to the customer and an adjustment notification to the seller saying that he has to change the state of the product to bound. In case, if this message fails the seller will still have an item in unbound state. In the next event if the seller receives more quantity (let's say 3) of product A, he will try to update the core's image with 4 as unbound quantity, but the actual unbounded orders are 3. Thus causing the buyer of the last item of this product to get an order successful message even though the seller does not have the product to deliver. This type of discrepancy has to be resolved as soon as possible to avoid such situations.

This can be solved by verifying both the images and raising an alarm if any discrepancy was found. So, we need to run a script which will generate a snapshot of the inventory image and send it to core, so that it will validate both the images. Typical large warehouse contains 1,00,000 different products and the inventory image contains different IDs for a product and quantity in all the dispositions. Each product data looks as below:

```
 WAREHOUSE            CORE      <quantity in dispositions> . ..
 Internal Id         Item Id

dellxps4Z2dd2,  laptopr24r3gu5 ,12,3,0,0,0,3,0,0,0,0
smsung8beAd23,  laptops5yg3efg ,10,0,0,1,0,0,0,0,0,4
rcChoppr8e3eq,  toyrca4Ur3r2gb ,45,0,0,1,0,4,0,0,2,0
woodendsk2da2,  furnitd3BGLft4 ,32,0,0,0,0,0,0,0,0,0
watrbottl7sw8,  bottleda98salL ,42,0,0,0,2,0,0,0,5,0
```

Figure 1.4: shows inventory snapshot image of a warehouse

In the above Fig. 1.4, there are multiple columns, first two being SKU Ids, while the others are the available quantities in each disposition. Few examples of the quantity dispositions are AVAILABLE, BOUND, DAMAGED and etc..

Max size of each item data can be 100 chars, i.e., 100bytes. If there are 100000 such items the raw data file will be up to 10MB. But if the same is marshaled to markup language can expand to size of 400MB. In reality one such xml shall be created for each and every warehouse probably in a single host. So, there is a strong need of stream marshalling instead of converting the entire message in memory.

1.2 Objective

Given raw image of inventory, we have to figure out a way for marshaling the raw data into a transmission message, which essentially is in xml format. Script, which implements this functionality should also parallelly upload the marshalled data into two buckets of S3 (an Amazon storage service), one, named backup Bucket for backing up data and other, named outboundBucket for sharing the image to core service. OutboundBucket needs the data inside it to be encrypted for security purpose.

Figure 1.5 shows the flow of marshalled data

The arrow marks in the Fig. 1.5 represents stream I/O operations.
The raw CSV file is stream read by the marshalling algorithm. The input will be eventually marshalled and written as streams into the storage.

The above Fig. 1.5 shows the flow of the data. The data after marshalling must be backed up and also is stored in a bucket with encryption enabled. The encryption is essential because the message contains critical data, and the backup message is non-encrypted because the content has to be readable for the technicians who may have to manually process message in case of failures.

Chapter 2
Literature Survey

The project aims to design and develop a scalable algorithm to create messages in data-transfer format from massive datasets. In this literature review, we will discuss on the current state of xml marshalling practices and etc. We also employ necessary logical procedures to make the algorithm encrypt message on-the-go. So, this discussion contains citations of the corresponding symmetric key encryption algorithms. Along with the implementation, we planned to validate the generated message with its schema. And hence, we also cite xml validation techniques in this survey.

2.1 Sun Microsystems, Inc's JAXB Architecture

Joseph Fialli & Sekhar Vajjhala (January, 2003) [1] has developed a Java Architecture for XML Binding also known as JAXB Specification. JAXB is a software framework that allows Java developers to map Java classes to XML representations. JAXB provides two main features: the ability to marshal Java objects into XML and the inverse, i.e. to un-marshal XML back into Java objects. In other words, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure.

The XML marshaller's flow architecture is as described in the following Fig.

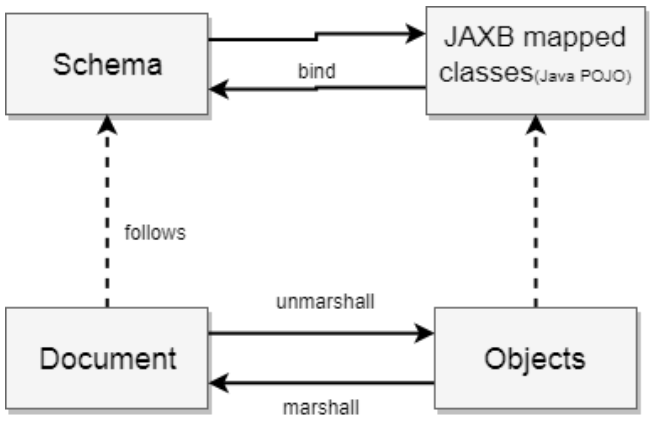


Figure 2.1: JAXB binding process

In the figure Fig.2.1 , we can see that document can be unmarshalled to create a memory Object. And Vice versa by marshaling. The bind process means, The created JAXB POJOs will be bound to the XML schema. And XML document will be created following that schema.

The Fig. 2.1 describes the binding/unbinding processes performed by JAXB for marshalling the data.

When this process is applied to convert an XML document to an object, it is called *un-marshalling*. The reverse process, to serialize an object as XML, is called *marshalling*.

Currently StAX architecture built by Haustein S. and Slominski A. [2] provides functionality to stream convert object to xml, but if we consider the case of converting directly to encrypted data, we will find difficulties because, the output stream should always provide a string of length which should be a multiple of cipher length. Few more constraints along with this have to be considered while solving our issue.

2.2 AES algorithm

In this paper we make use of 128-bit Advanced Encryption Standard (AES) symmetric key encryption algorithm created by two Belgian computer scientists, Vincent Rijmen and Joan Daemen [3]. We use this to serve the purpose of protecting the output data. We will integrate this encryption algorithm in our implementation so that the stream will generate encrypted data.

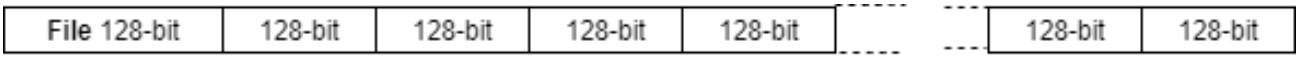


Figure 2.2: file fragmented for encryption

For the total file encryption, the algorithm will be applied on the chunks (each 128-bit) of the file.

2.3 XML Validations

In this report, we will be using XSD schema validation technique [4] to prove the authentication of the stream created XML file. XSD (XML Schema Definition), a recommendation of the World Wide Web Consortium (W3C), specifies how to formally describe the elements in an XML document. It can be used to verify each piece of item content in a document. This schema helps to verify if a document adheres to the description of the element it is placed in.

Chapter 3

Stream Marshalling

3.1 Overview

From the Fig. 3.1 we can observe that the large size of xml is due to the repeated element, which is storing data about the availability details of each item (element named as <inventoryItemData>...</inventoryItemData>).

```
<? xml version="1.0" encoding="UTF-8"?>
    <transmission id="3244252223" >
        <message id = "5423" messageType="IIS">
            <InventoryImageNotification>
                    <InventoryControlDetail time="Sun, 08 Oct 2017 12:46:06 GMT"/>

                    <InventoryItemData>...</InventoryItemData>
                    <InventoryItemData>...</InventoryItemData>
                    <InventoryItemData>...</InventoryItemData>
                    <InventoryItemData>...</InventoryItemData>
                    <InventoryItemData>...</InventoryItemData>
                    <InventoryItemData>...</InventoryItemData>
                    . .
                    .
                    .
            </InventoryImageNotification>
    </message>
</transmission>
```

Figure 3.1: Structure of the marshalled data

This Fig. 3.1 contains an example of the marshalled invenotry snapshot image, which serves the purpose of syncing two inventory snapshots. The data is marshalled into a transmission, which in turn contains a sycn message called InventoryImageNotification.

If we have to stream output the result we should be able to divide the output in to multiple chunks, but for each chunk, we should be able create XML from the given information/raw data.

We made use of streams to overcome the memory limitations imposed by the previous algorithm. In order actually not use much memory; we need to write/upload the partially marshalled XML data to a local file/storage service. Thus not keeping the total marshalled XML, i.e. output in the memory.

As per the application's requirement I chose to upload the data into Amazon's S3 service. But there were few requirements for the usage of the service. The requirements are as follows:

a. The size of the each fragment/chunk which has to be uploaded must not be less than 5MB in size. Except the last part can have any size.

Why should the size be more than 5MB?

The storage service has kept this constraint to avoid unnecessary network calls trying to upload very small data chunks. Thus, efficiency of each fragmented upload.

b. All the segments should be uploaded in the correct order. Since the service keeps on appending the data sequentially any change in the order of segments will corrupt the data

c. In case of server side encryption (SSE) upload, each fragment uploaded must have its length as a multiple of the length of the cipher key used in the encryption. That is, if the length of the cipher key is 128-bit, then the each fragment can only take sizes of 128 bits, 256 bits, 384 bits.. or (128)*m for any $m \in \{1,2,3\ldots.\}$

Reason behind the constraint:

First of all, we need to know how a file is encrypted in our case. In the Fig. 2.2 we saw that the file is broken in to segments each of the length of the cipher key used for encryption. Then encryption is performed on each of them.

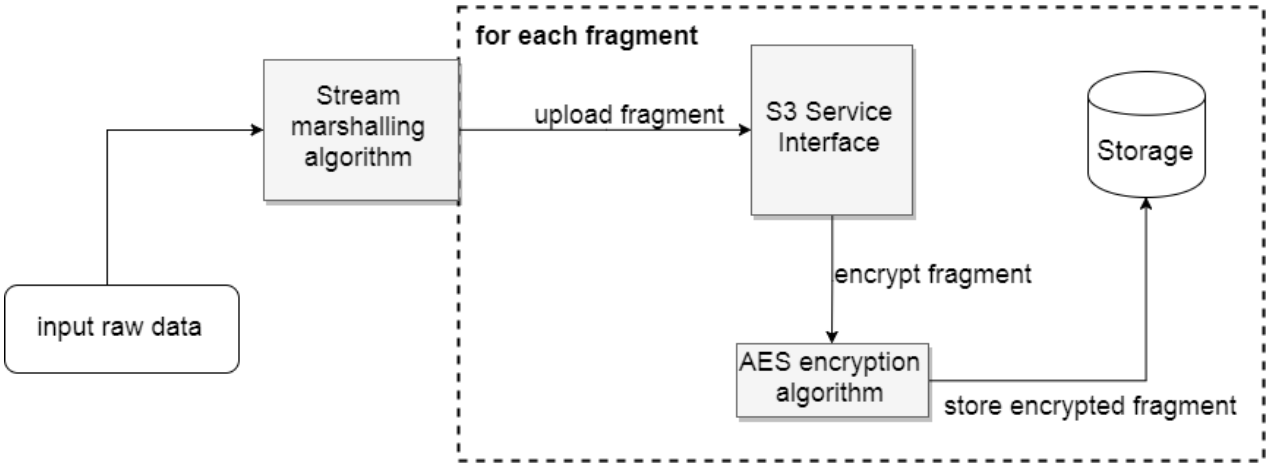


Figure 3.2: describes the flow of control for encrypting file

The design / flow of control diagram shown in the Fig. 3.2 corresponds to the multi part uploading of a file to S3 storage service, with SSE (Server Side Encryption) enabled.. When encryption is applied on a segment which is less than the size of the cipher key, the encryption will be performed after increasing the segment length to the length of cipher key by appending zeros (binary) to the segment. So, whenever the encrypted segment is decrypted it will produce a data having zeros at the end.

So, if any fragment whose length is not a multiple of the length of the cipher key is uploaded, the service will try to encrypt it by breaking them into segments of 128-bit, but the last segment will be partial and zeros will be appended. Since, the upload might not be the final one, unnecessary zeros will be present inside the data, thus corrupting it.

Thus to protect the client from any accidental data corruptions, the storage service has imposed this constraint on the size of the each segment.

3.2 Architecture for streaming

In this section we will design the logic for marshalling the data using streams. We will be handling the constraints in the later section 3.3. To design architecture for stream creation of the XML file, first we need to inspect the structure of XML document.

From the Fig.3.1 we can observe that the XML message has a repeated element (InventoryItemData). And it has message headers which contain meta data and message footers corresponding to the message headers. These three parts can be divided as shown in the following Fig.3.3.

```
----------Part (1) ----------

<? xml version="1.0" encoding="UTF-8"?>
<transmission id="3244252223" >
    <message id = "5423" messageType="IIS">
        <InventoryImageNotification>
            <InventoryControlDetail time="Sun, 08 Oct 2017 12:46:06 GMT"/>
---------- Part (2) ----------
            <InventoryItemData>...</InventoryItemData>
            <InventoryItemData>...</InventoryItemData>
              . .
               .
               .
---------- Part (3) ----------
        </InventoryImageNotification>
    </message>
</transmission>
```

Figure 3.3: the three parts of the repeated large xml files, Part (1): Meta-data header Part(2): Repeated element (stream able part) Part(3): Meta-data footer

To marshall such an XML file directly using the existing frameworks, we create classes as per the xml structure and provide it to framework. The framework will then make use of this schema to convert the data into the corresponding XML document.

Example of the class structure:

To create an XML document as shown in Fig.3.3, we need classes to be implemented as shown in the following Fig. 3.4:

Class Structures

Transmission: {**Message**, Id}

Message: {**InventoryImageNotification**, Id, MessageType}

InventoryImageNotification: {**InventoryControlDetail**, List< **InventoryItemData** >}

Figure 3.4: class structures defining the XML schema

But if we want to marshall the XML in streams we have to first create the metadata Headers (part -1), then the stream able part (part -2) and finally metadata footers (part-3), every part referred in the Fig. 3.3.

To create the metadata without the repeated part (List<**InventoryItemData**> in our example), we will pass the Message object to the framework's API with the List<**InventoryItemData**> as null, thus creating the combined 1st and 3rd part. The marshalled data will be as shown in the Fig. 3.5

For the next step we will split the marshalled data to get part-1 and part-3. Now that we have part-1 we can upload it and delete it from the memory. Next, we will stream read each row from the input (.csv file) raw data, and for each row, we will marshall data using existing framework, and upload them. Finally, in the last step we will upload the part-3 to complete the marshalling.

```
<? xml version="1.0" encoding="UTF-8"?>
<transmission id="3298" >
      <message id = "5973" messageType="IIS">
            <InventoryImageNotification>
                  <InventoryControlDetail time="Sun, 08 Oct 2017 12:46:06 GMT"/>
----------<InventoryItemData/>---------- (element = null)
            </InventoryImageNotification>
      </message>
</transmission>
```

Figure 3.5 marshalled XML without repeated element (< InventoryItemData >)

3.2.1 Model architecture:

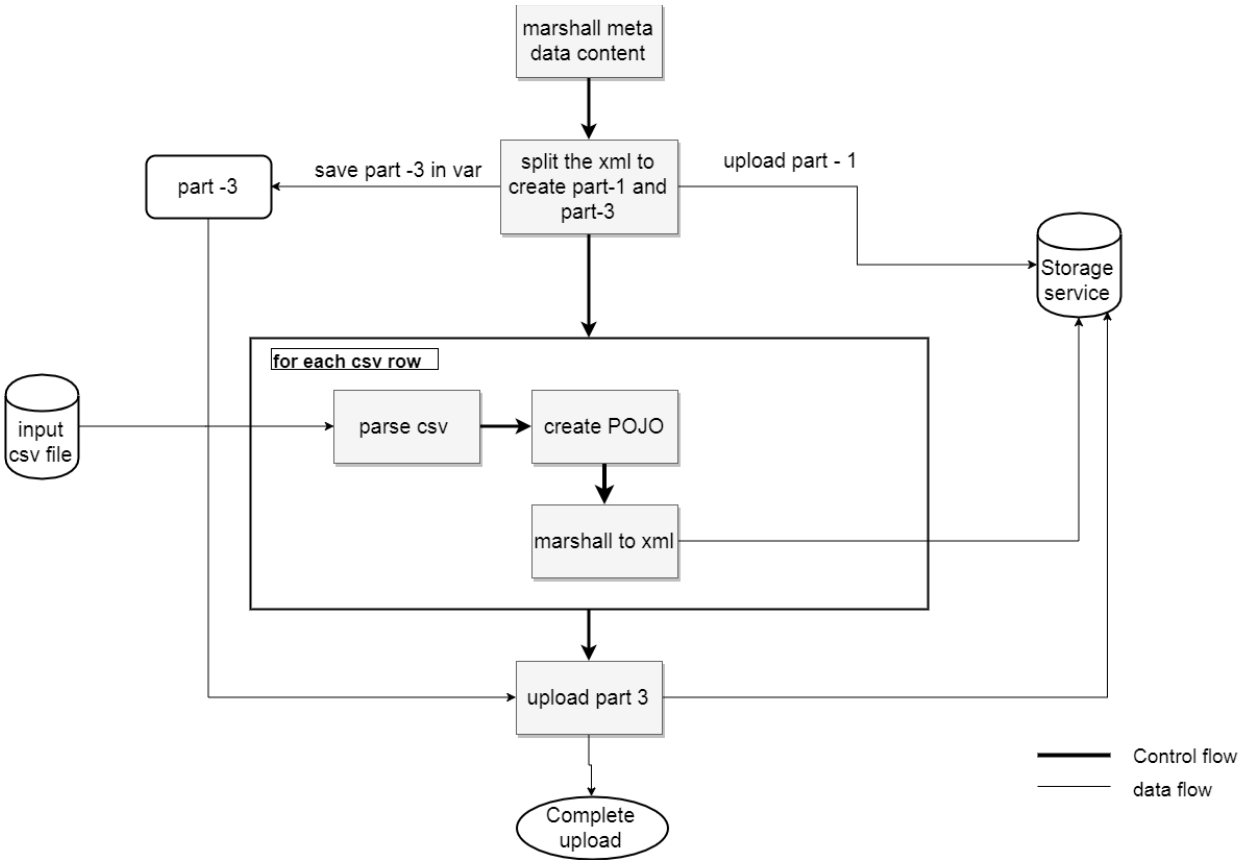


Figure 3.6: model of the architecture of stream marshalling

The dark line/thick line in the Fig. 3.6 represents the flow of control, while the weak line represents the flow of information/data between CPU RAM and the storage. The flow of control starts from marshalling of meta data content and ends at the Complete Upload.

3.3 Handling the constraints

The storage service's multipart upload API has imposed two significant constraints. The constraints are as follows:

a) Each upload fragment must have a minimum size of 5MB
b) Size of the upload fragment should be multiple of the size of the cipher key (constant = 128-bit)

From the above data, we can conclude that this is not as same as stream uploading, because the stream uploader will not have any such conditions.

But the stream marshalling algorithm which we have designed is trying to upload as a stream. The algorithm need not know/care about these conditions, as these are service specific.

To solve this problem I have deigned a masking solution, which will mask the multipart upload as stream upload.

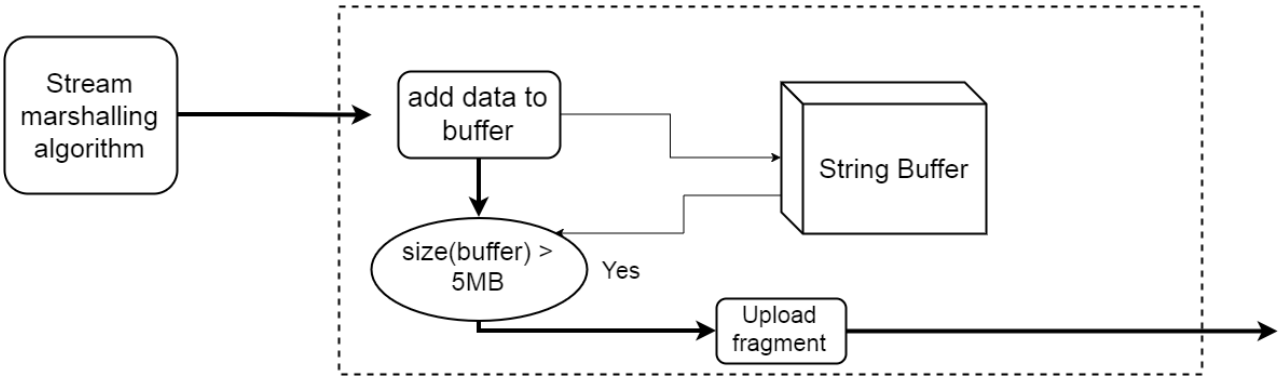


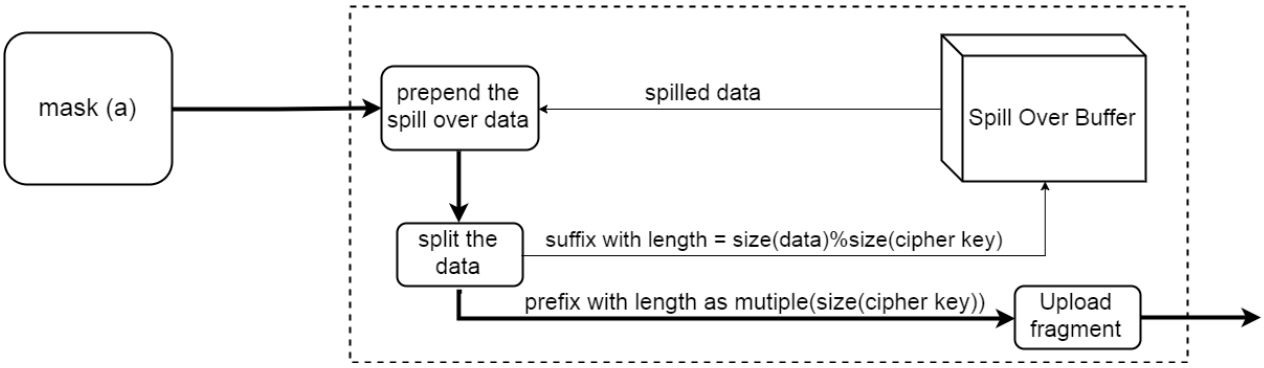
Figure 3.7: architecture for the multipart upload mask (masking the constraint - a)

The architecture of the mask (for making the first constraint a) is defined as mentioned in the Fig. 3.7.

Here the mask is a state-full class, maintaining a buffer, which stores all the fragments input to it and once the buffer fills, it will upload the whole as a fragment. Thus, the user of this mask does not need to concern about the minimum required size of the upload fragment.

Now the second constraint (b) is yet to be masked. To solve this we will add one more mask over the previously created mask (a). This will contain the mechanism to break the fragments for to create

chunks of length multiples of 128 bits. We will create a buffer called spill over buffer, to store all the parted chunks. The mechanism is as follows:



i. When a data is given as input to the mask, the previously spilled data is prepended to it

Figure 3.8 architecture for the multipart upload mask- b (masking the constraint - b)

ii. This added data of size **α** will then be split into two parts,

First part is the prefix with the size as nearest multiple of e, where e = size of the cipher key

First part of the data = data from 1^{st} bit to $(\boldsymbol{\alpha} - \boldsymbol{\alpha} \% e)^{th}$ bit

Second part = data from $(\boldsymbol{\alpha} - \boldsymbol{\alpha} \% e + 1)^{th}$ bit to last bit

iii. Second part will be stored in the spill over buffer, while the first part is considered as the output

Corner cases like the uploading of the last fragment will also be taken care of by adding a flag which will turn true if the fragment is the last one.

The Complete Architecture Model

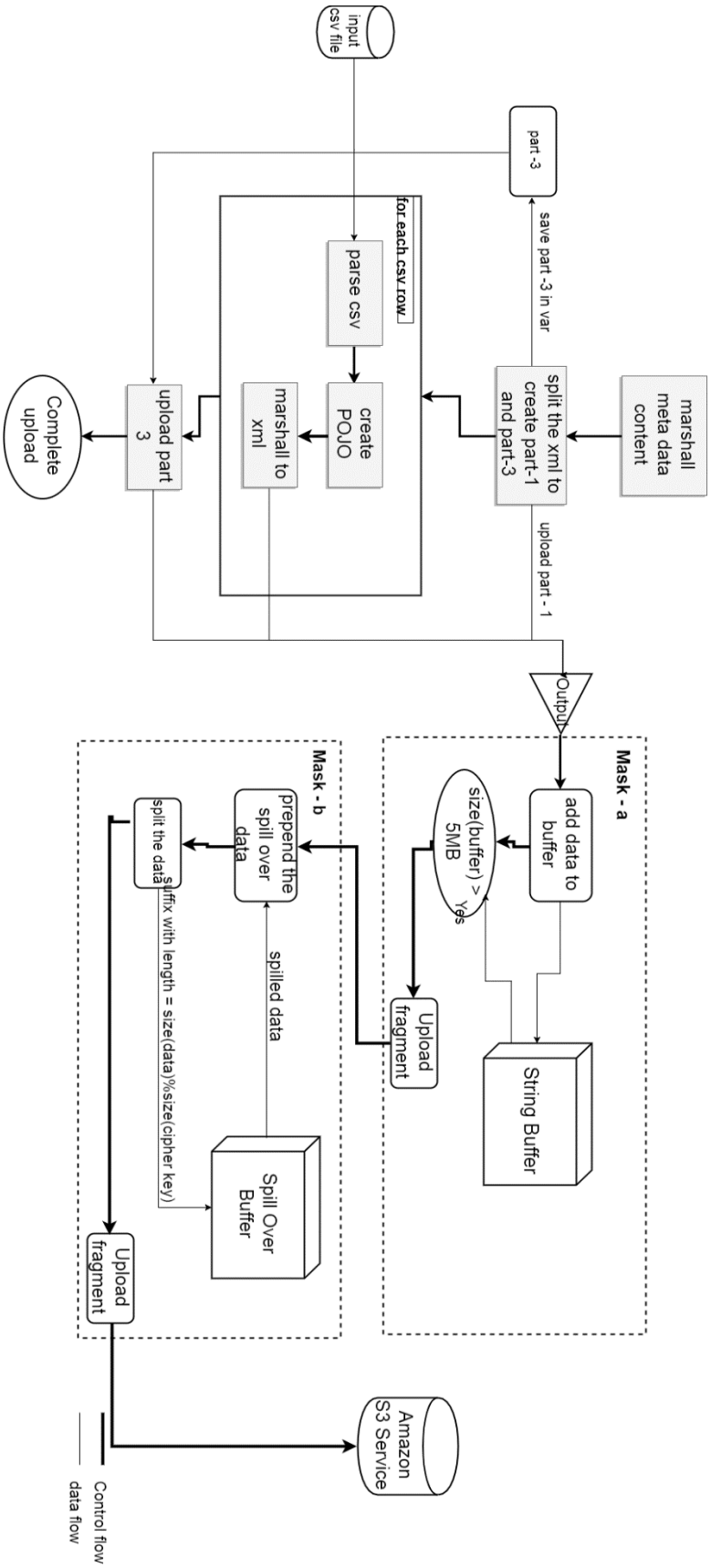


Figure 3.9 The complete architecture of the stream marshalling algorithm

Chapter 4
Experimental Walkthrough and Results

All the tests were performed on a x86 architecture 64 bit **m4.large** Linux machine having processor specification as: 2.3 GHz Intel Xeon® E5-2686 v4 (Broadwell) processor with 8GB RAM, SSD storage and 450Mbps Dedicated EBS bandwidth.

4.1 Generated XML Validation testing

Now that we have implemented the script which stream generates the XML document from the raw csv data, we need to check if the output is still a valid XML document.

For validating the generated XML (refer Fig. 3.1), here we use XSD language. The XSD we have used for the validation is as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xs:element name="transmission">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="message" />
   </xs:sequence>
   <xs:attribute name="id" use="required" type="xs:string" />
  </xs:complexType>
 </xs:element>
 <xs:element name="message">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="InventoryLevelsNotification" />
   </xs:sequence>
   <xs:attribute name="messageType" use="required" type="xs:string" />
   <xs:attribute name="id" use="required" type="xs:string" />
  </xs:complexType>
 </xs:element>
 <xs:element name="InventoryLevelsNotification">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="inventoryControlDetail" />
    <xs:element maxOccurs="unbounded" ref="inventoryItemData" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="inventoryControlDetail">
  <xs:complexType>
   <xs:attribute name="time" use="required" type="xs:string" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 4.1 XSD schema for validating the marshalled content

This XSD contains same structure information which is also defined by the classes in the Fig. 3.4. But this XSD schema also stores information about the data types. The generated XML **has passed** this schema validation. Thus, we can confirm that the implementation does not have any regressions.

For the raw-data:

samNote8S, note8A5ty, 5, 0, 0, 5

xpsDrone89, drone9Iu0S, 5, 0, 5, 0

This input contains two Stock Keeping Units, with two Ids each along with quantities in Four different disposition

The generated XML data (displayed in the Fig. 4.2) is:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<transmission id="24309756434">
  <message id="342424">
    <InventoryImageNotification>
      <inventoryControlDetail inventoryControlType="FULL" />
      <inventoryItemData>
        <itemID type="AMAZON_ASIN">samNote8S</itemID>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">5</quantity>
          </itemQuantity>
          <itemLocation>RECEIVE</itemLocation>
        </availabilityDetail>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">0</quantity>
          </itemQuantity>
          <itemLocation>PRIME</itemLocation>
        </availabilityDetail>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">0</quantity>
          </itemQuantity>
          <itemLocation>CORDER-COMMITTED</itemLocation>
        </availabilityDetail>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">5</quantity>
          </itemQuantity>
          <itemLocation>CUSTOMER-RETURN</itemLocation>
        </availabilityDetail>
      </inventoryItemData>
      <inventoryItemData>
        <itemID type="AMAZON_ASIN"> xpsDrone89</itemID>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">5</quantity>
          </itemQuantity>
          <itemLocation>RECEIVE</itemLocation>
        </availabilityDetail>
        <availabilityDetail>
          <itemQuantity>
            <quantity unitOfMeasure="EA">0</quantity>
          </itemQuantity>
```

```
        <itemLocation>PRIME</itemLocation>
      </availabilityDetail>
      <availabilityDetail>
        <itemQuantity>
          <quantity unitOfMeasure="EA">5</quantity>
        </itemQuantity>
        <itemLocation>CORDER-COMMITTED</itemLocation>
      </availabilityDetail>
      <availabilityDetail>
        <itemQuantity>
          <quantity unitOfMeasure="EA">0</quantity>
        </itemQuantity>
        <itemLocation>CUSTOMER-RETURN</itemLocation>
      </availabilityDetail>
    </inventoryItemData>
  </InventoryImageNotification>
 </message>
</transmission>
```

Figure 4.2 The sample marshalled Data

In the above test, the input is so small that our code will upload it in a single fragment. Here, we actually did not test if the code works for large data. To test the code for multiple fragments, we can create a large xml with at least 10k items. So, our next text will be having 70000 items in raw input.

Output is an xml file [5] with size 140MB created as 29 different fragments from the input. This file is then finally validated on the schema and results were positive.

Next, we have to test if the algorithm is efficiently running in a limited memory environment. For this test we will be comparing our algorithm's efficiency with general marshalling algorithm.

4.2 Algorithm performance testing

Firstly, we will compare both the algorithms by running them on relatively **small data**.

Input data: 4 random csv file with moderate number of rows (200 rows) are simulated

Times taken for the algorithms are noted down as follows:

Table 4.1 execution time taken for both the algorithms on moderate data set

Dataset	JAXB framework	Stream Marshalling
1	443ms	424ms
2	430ms	451ms
3	449ms	435ms
4	434ms	440ms
Average	439ms	437.5ms

From the results shown in Table 4.1, it is unclear about the execution time efficiency of the algorithms, since the results contain very similar results. Also the generated XML file was very small (in kilo bytes) for any segmented uploads to happen. So we need to test on larger data set to confirm the behaviors.

Input data: This time, 4 random csv files with very large number of rows (2,00,000 rows) are simulated

Times taken for the algorithms are noted down as follows:

Table 4.2 execution time taken for both the algorithms on very large data set

Dataset	JAXB framework	Stream Marshalling
1	472.439sec	433.587sec
2	469.827sec	430.544sec
3	480.251sec	442.287sec
4	475.672sec	440.996sec
Average	**474.547sec**	**436.85sec**

From the results in Table 4.2 it is evident that stream marshalling algorithm is faster than JAXB framework.

The **36 sec** advance of the stream marshaller over JAXB can be explained from the two factors:

a) Since JAXB[10] stores all the marshalled data in memory, eventually the host runs out of memory and the system might have forced the application to use the swap memory, which will drastically decrease the performance. Also if the memory is bound, it will be harder to gather free memory and use it.

b) In the case of JAXB, the upload was done after the completion of marshalling. Thus system has no choice other than sequentially marshalling and then uploading. But in case of stream marshalling, CPU will be busy marshalling the data, while upload will be done in parallel, because for uploading DMA will be used. Thus taking advantage of DMA[11], stream marshalling will be faster than the general marshalling.

4.3 Algorithm's memory efficiency

From the above tests we can conclude that stream marshaller is faster than the general marshaller. Next, we will be testing the memory consumption of both the algorithms with time. The result was as shown in the Table 4.3

Table 4.3 memory-time relation data for both JAXB and stream marshall algorithms

Time(t)	**Memory consumed**	
	JAXB	Stream marshaller
10sec	15.4MB	7.9MB
60sec	89.3MB	8.0MB
110sec	168.0MB	8.1MB
160sec	244.3MB	8.1MB
210sec	321.2MB	8.1MB
260sec	393.5MB	6.7MB
310sec	462.8MB	8.1MB
360sec	528.7MB	5.2MB
410sec	586.2MB	8.1MB

The memory consumed by the program is calculated for every 50 sec, by the use of 'smem' command. The procedure is done for both the existing algorithm and our algorithm. The data collected is as shown in the Table 4.3.

We can observe that memory consumed by JAXB is increasing nearly at the rate of 7.61MB for every 5 seconds. For our algorithm, the value is almost constant.

We can see some spikes at 260^{th} second and 360^{th} second, these spikes can be explained. In the algorithm we can see that there is an instant where the existing data (8.1 MB in the example) is uploaded and the script is marshaling the next fragment. In this period, the memory consumed will be different from the constant memory observed.

While collecting the data, it was also observed that, JAXB was taking a maximum of 612.2MB at one instant of time, while Stream marshaller was only consuming 8.4MB at it peak stage.

This data was collected using a simple bash script which uses ***smem*** command to get memory details. The script collects the memory consumption details of the program for every 50 seconds.

These details in table 4.2 can be easily visualized under a graph. After plotting in the graph, the results were:

For JAXB execution:

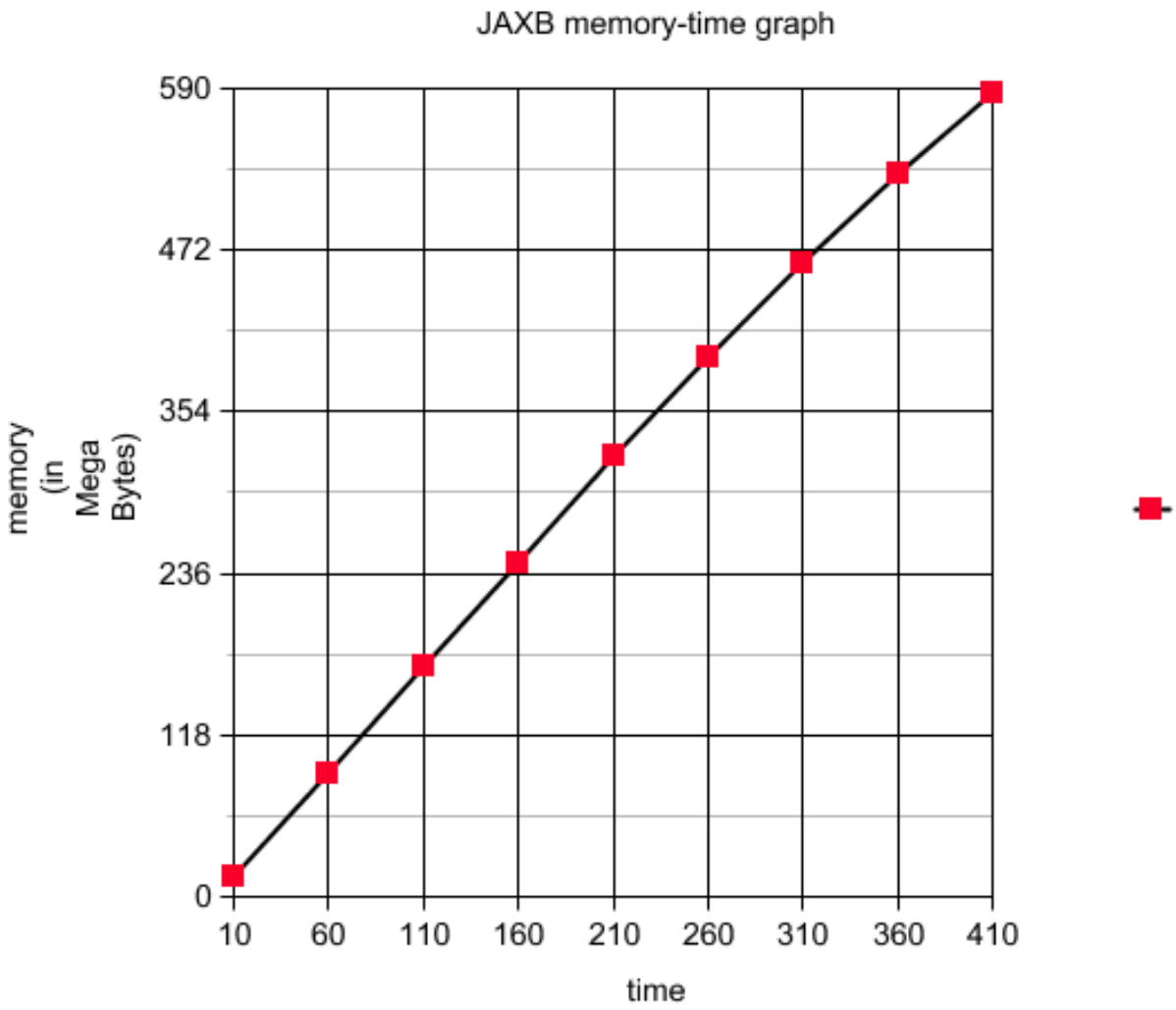


Figure 4.3 memory-time relation graph for JAXB execution on very large data set

The graph shows that the memory consumed was increasing almost linearly with time. This is essentially because the marshalled data gets piled up in the memory.

The x-axis of the graph shown in Fig. 4.3 represents time (during the execution of the algorithm), values starting from 0, which is the start if the execution time.

In the y-axis, we plot the values of memory-consumption by the program. Each data point, which essentially is a tuple containing a point of time and a value of consumed memory, represents the memory consumed by the program at mentioned time since it has started its execution.

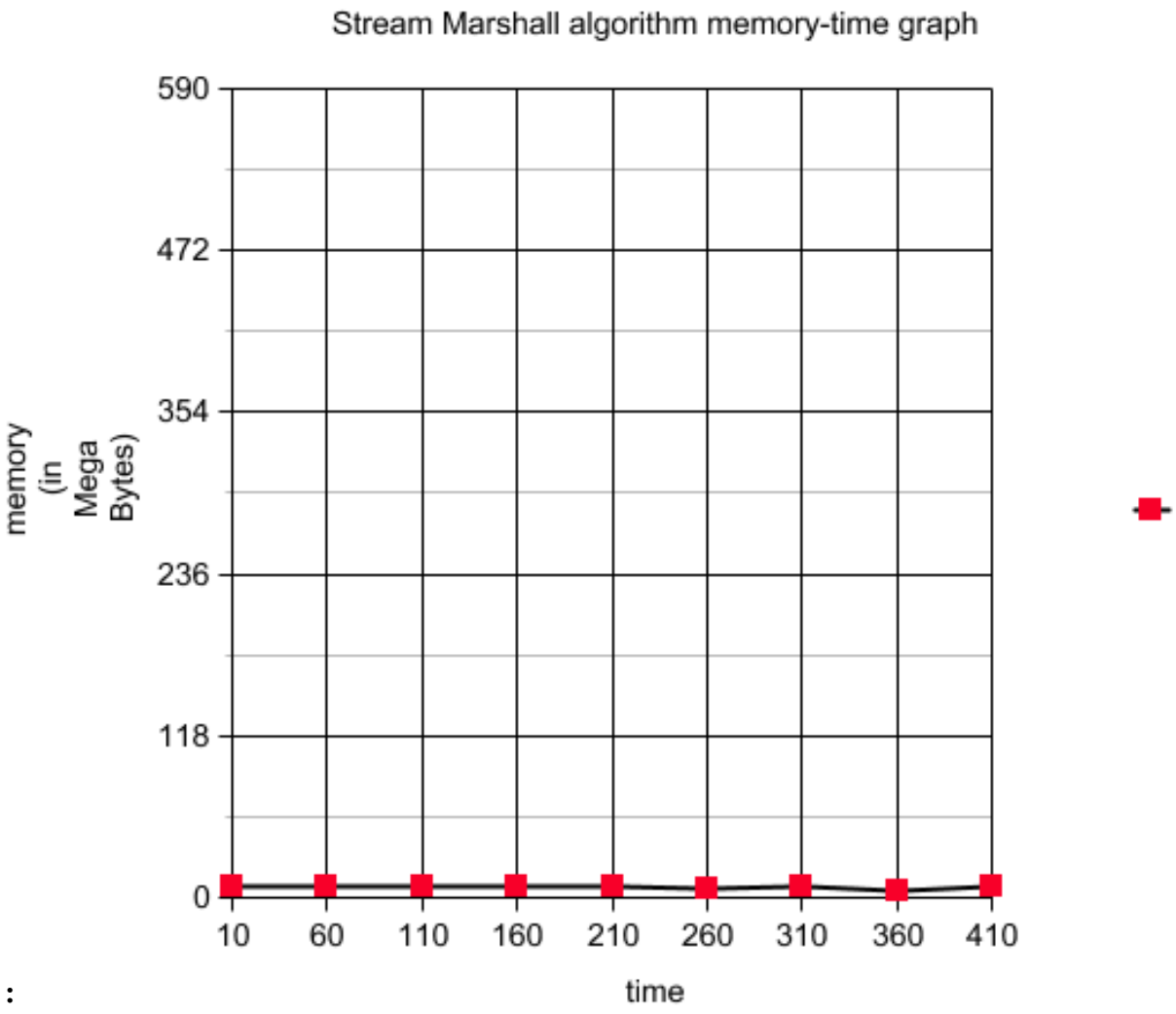
The squared dots shown on the Fig. 4.3 represents the collected data points, and the black line is the interpolation of the memory-consumption at remaining points of time.

We can clearly notice that the greatest disadvantage of JAXB framework is that the memory consumed is growing up drastically. In the above case of Fig. 4.3, it is observed to have a slope of 1.522 Mbytes/ 1second.

Even though todays have large memory attached to them, memory consumption is not a desired property. Improving this would be always beneficial, because

a) The greater the memory consumption is, the greater will be the probability of swap memory being used for the execution.
b) The greater the memory consumption is, the greater will be the access times.
c) Also, this will limit the execution of multiple instances of the algorithm in the same host

For Stream marshalling execution



:

Figure 4.4 memory-time relation graph for stream marshall algorithm execution on very large data set

Even for this graph, the x-axis of the graph shown in Fig. 4.4 represents time (during the execution of the algorithm), values starting from 0, which is the start if the execution time.

In the y-axis, we plot the values of memory-consumption by the program. Each data point, which essentially is a tuple containing a point of time and a value of consumed memory, represents the memory consumed by the program at mentioned time since it has started its execution.

Unlike the previous graph this is only taking almost constant value all the time. This is because the algorithm is uploading the data as it was marshalled, thus not keeping any data in memory. This helped the algorithm to use much less memory.

Hence this algorithm can have multiple instances parallely running in a single host, without ever facing out of memory error.

All the results were observed exactly as expected. Also the implementation of this algorithm is put into real use for marshalling inventory images of seller warehouses in Amazon. Many metrics were added to the script, so that any unexpected behavior triggers an alarm. As of now no such alarms were raised.

Chapter 5

Conclusion and Future Work

Conclusion

Now we have successfully created a big xml file (size over giga-bytes) that was marshalled from a large input. Fragmentation has an advantage over marshalling large files. Also maintaining a spill over buffer allows us to fragment over a constraint that the size should be a multiple of length of cipher key. Now on, we will be able to marshall large files rapidly.

Future Work

The future work of this project includes the creation of a framework, which marshalls data in streams. Current frameworks which can marshall as a stream provide limited functionality. They are only able to save the data to a file and they give no control over fragmentation. Our framework will be provided with a buffer reader to read input raw data and a converter class to know the logic of converting csv row to memory object.

The framework works like a stream converter, which converts input csv file reader to XML document streamer. That is, the input of this framework will be a stream and also the output will be a stream, except that the output stream directly writes marshalled data. The framework will also provide essential tools for uploading data with constraints.

Finally the usage of the frame work should look as simple as:

```
streamMarshaller.setReader(reader);
streamMarshaller.setMinSize(5*1024*1024); // 5mb in bytes
streamMarshaller.setCipherLength (128); // in bits
streamMarshaller.setConverter(new RawDataHandler(){
                public Object convert(String inputFragment){
                        // our parser implementation
                        return object;
                }
});
while(!streamMarshaller.completed()){
        upload(streamMarshaller.getFragment());
}
```

To implement the script on the proposed algorithm, I had to write 3500+ lines of code. This is actually because of the logical implementation being needed to take care of. With the above proposed framework can drastically improve the implementation of such algorithms. Also, the number of lines of code needed for the implementation would be decreased to about 400.

References

[1] Fialli, J., & Vajjhala, S. (2003). The Java architecture for XML binding (JAXB). JSR, JCP, January

[2] L. Segoufin, V. Vianu: Validating Streaming XML Documents, PODS, 2002.

[3] Bellare, M., Desai, A., Jokipii, E. and P. Rogaway, "A Concrete Treatment of Symmetric Encryption: Analysis of DES Modes of Operation", Proceedings 38th IEEE FOCS, pp. 394- 403, 1997.

[4] Andrew Eisenberg, IBM, Westford, MA & Jim Melton, Oracle Corp., Sandy, UT. SQL/XML is making good progress, June 2002

[5] https://goo.gl/gzSDhH

[6] https://www.w3schools.com/xml/schema_intro.asp

[7] https://www.investopedia.com/terms/s/stock-keeping-unit-sku.asp

[8] https://services.amazon.in/services/sell-on-amazon

[9] https://aws.amazon.com/s3/

[10] http://www.oracle.com/technetwork/articles/javase/index-140168.html

[11] https://www.techopedia.com/definition/2767/direct-memory-access-dma