IMAGE PROCESSING ON A DEPENDABLE DISTRIBUTED SYSTEM USING RMI

Final Project Report

Georgia Institute of Technology
ECE 6102 Dependable Distributed Systems
Spring 2016

Team Members:
Prapurna Jayakrishna Duvvuri (GTID: 903091962)
Uday Kiran Ravuri (GTID: 903145978)
Mrunmayi Sharad Paranjape (GTID: 903117020)
Rudra Dushyant Purohit (GTID: 903123137)
Ankul Jain (GTID: 903045020)

Abstract

This project uses the robust Java RMI (Remote Method Invocation) to implement a global snapshot technique using the Chandy Lamport algorithm as described in the paper - 'Distributed Snapshots: Determining Global States of Distributed Systems' [1] and demonstrate the process flow using a dependable distributed system wherein all nodes run the same image processing application. The dependability is achieved through checkpointing using global snapshots.

I. INTRODUCTION

1.1 Motivation

Distributed systems are gaining wide importance in processing parallelizable tasks simultaneously. Lot of computation work can be distributed amongst various nodes. This helps in proper load balancing. However, the distributed systems face problems of inconsistent data, instability and failure in case of a node breakdown. To solve the problem of recovery, the Chandy Lamport paper proposes an algorithm where the global state of all the processes is saved periodically. We aim to develop a system based on this algorithm, and demonstrate the stability and fault tolerance achieved by implementing a simple image processing application.

II. PROJECT OVERVIEW

2.1 Services implemented

Application service: Distributed image processing application (explained in detail later)
Reliability services: Dependability, Consistency, Replication, Parallelism

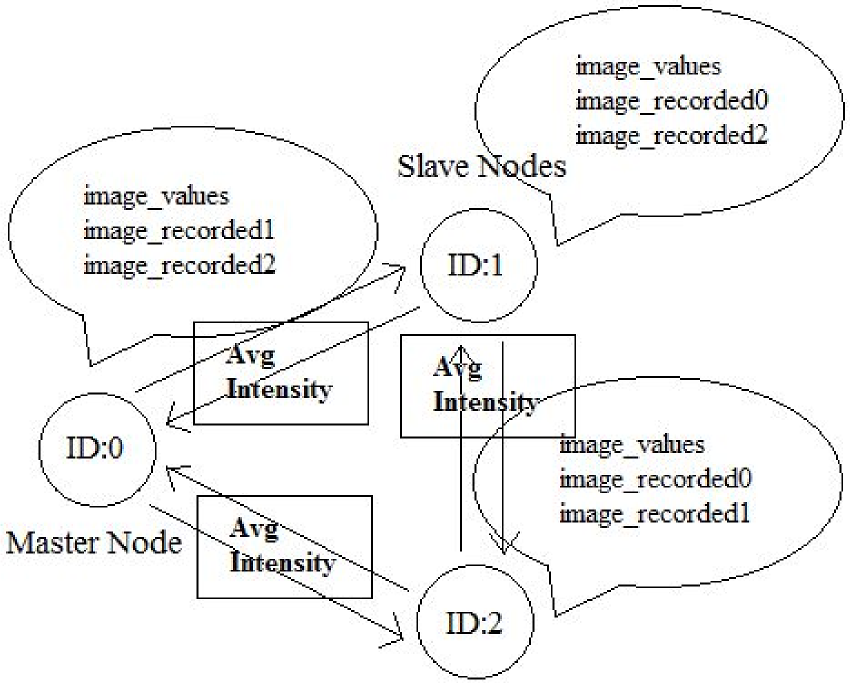


Fig. 1: Block Diagram - High level diagram showing processes' communication flows

2.2 High-level description

Three processes running on three nodes, which run the same distributed application code. The application implements an image processing solution to control the brightness of monitors. To achieve the proposed service requirements, they communicate by sending/receiving messages. These messages contain the mean intensity values of images. With this as the underlying process, one of the nodes initiates the snapshot algorithm (Chandy Lamport) which captures the global states of all processes and channels between them. The latter is achieved using marker messages as mentioned in the paper. The snapshot thread terminates when all processes receive markers and all channel states have been recorded, although the application messages continue flowing through the network. Using this snapshot, we demonstrate the stability of the system by reaching the same end state after killing and restarting the processes in all nodes. We achieve dependability and fault tolerance by replicating data among the other two processes.

2.3 Facilities

Hardware

We used three laptops with unique IDs (IP Addresses in this case) as three nodes in our distributed system. To create and stabilize the communication, we had to:

- Connect to the same network
- Disable host firewalls
- Disable other running network adapters (other than the main WiFi connection)

Software

- Our code, including the image processing application and the algorithm implementation
- Image database (300 images on each node)
- Compiling .java files using Java 7 SDK
- We choose one of the nodes as the snapshot initiator, which has information about all the process and the channel states
- Every node's local storage has 3 files - image_values, image_recordedx, image_recordedy which are used to store the process and channel states. The values recorded in these files are used to evaluate our features, which is done in the sections below.

III. IMPLEMENTATION

To demonstrate the working of Chandy Lamport algorithm, we developed an interface which could be used by distributed application like Image Processing. The basic requirement to implement the algorithm was a distributed memory system with support for FIFO message passing for communication of markers and messages between the nodes. The high level description of our implementation is as follows.

3.1 Communication API

- RMI [2] provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.
- Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.
- This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application.
- The RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.
- RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine.

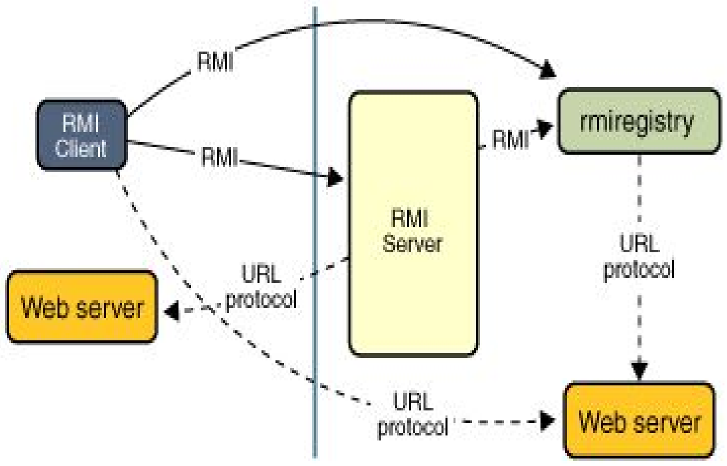


Fig. 2: RMI Block Diagram [2]

3.2 Image Processing application

Our main idea for this application is to be able to adjust the brightness level of the monitor screen using the average intensity values of the images. To this end, we chose 300 images [5] with varying properties like contrast, pixel values etc. Each process (3 in our case) chooses 1 set of 100 images based on its unique ID value and calculates the mean intensity value [4] of each and transmits it along its two outgoing channels to two other processes.

The processes stop sending messages when each node has intensity values of all 300 images in their local storages. The application messages continue to flow without any interruption from the marker messages received along the incoming channels as an indication to capture the process state. This feature is similar to what was described in the paper.

3.3 The Chandy Lamport Algorithm

The snapshot algorithm is an algorithm used in distributed systems for recording a consistent global state of an asynchronous system. It is based on a few assumptions that all the messages in a distributed system are eventually delivered in a FIFO manner. It also requires that the snapshot algorithm execution doesn't interfere with the normal execution of the system. The proper functioning of the Chandy Lamport Algorithm requires point to point communication between all the nodes in the system. It uses marker messages, which act as a synchronization primitive for taking snapshots.

The algorithm can be described as follows:

1. Any node in the system can act as the Initiator. The initiator process P_i records its own state. It then creates the special marker messages and sends it along all the outgoing channels. After sending out markers, it begins recording incoming messages on those channels.
2. Any other node, on receiving the maker message along a channel, reacts differently depending if it's the first marker that it is seeing.
   A. If it is the first marker P_j is seeing, P_j will first record its own state. It marks the state of the channel C_{ji} as empty.
   B. It sends out markers to all the nodes over all outgoing channels except to the node P_i
   C. It starts recording messages on each incoming channel at P_j
3. It the node P_j has already seen a marker message before, it marks the state of the channel C_{ki} as all the messages it received along it.
4. There are two necessary conditions for the algorithm to terminate:
   A. All processes have received a marker and have recorded their own state
   B. All processes have received a marker on all (N-1) incoming channels and recorded the state of all channels

3.4 Deviation from the initial proposal

1. We did not implement our code on the nodes in the Jinx cluster, as we had previously proposed, as we had limited access to these resources. Also, we needed to know their IP addresses for remote communication, which were not available to us. However, the fact that our code works perfectly on three different PCs with similar configurations implies that it can run on any other platform, given that Java is a platform-independent programming language.
2. We used the RMI interface rather than MPI [3] for communication. This is due to the fact that we encountered a problem with capturing the global state using MPI methods. Saving of system state on each process was achieved using MPI but the co-ordinated checkpointing in a non blocking manner (so that the application can remain in a running state) at the system level was tough to achieve using the semantics of MPI. We found our solution to this problem in Java RMI which provides us with a better tool for communication among processes.

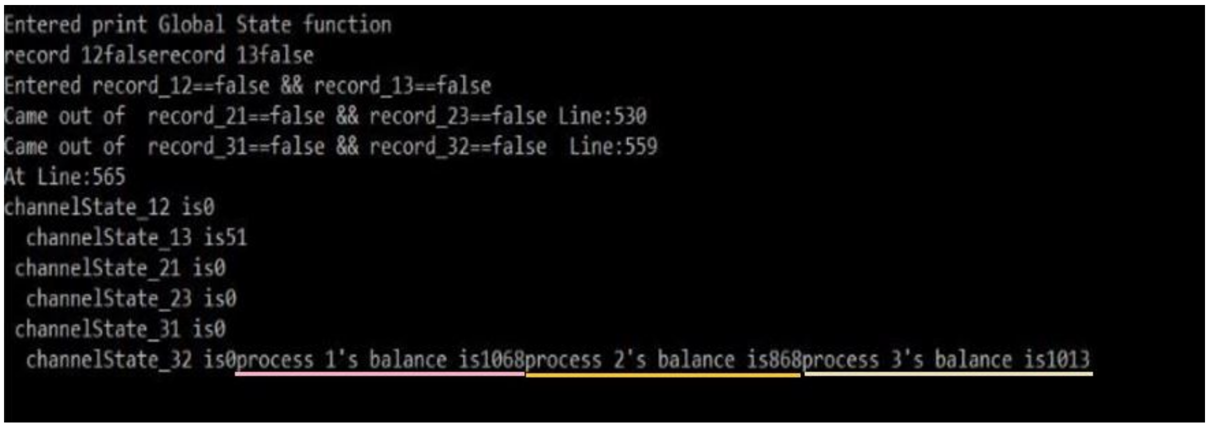
IV. EVALUATION

We demonstrated the validity of our implementation by testing the following features:

1. Dependability and Consistency - our system can restart from the latest checkpointed state stored in persistent memory and still can attain the same final state
2. Fault tolerance - Each image data (average intensity) is sent to two other processes in the system so that even when one node crashes, the other two have the shared data.

These features are explained through the console snapshots taken from all the nodes in our experiment:

1.
We first experimented by complete implementation of Chandy Lamport Algorithm for basic money transfer application:
We initialized each Process with a fixed amount of 1000, and during each message passing event, some amount was transferred to other node. The amount being transferred was deducted from the transmitting node's balance and was added to the receiving node's balance. Fig.3 below shows the global snapshot taken when the messages were being among the nodes.If the balances of each process and their respective channel state balances are added up, they add up to 3000, which is total money with which the three nodes started with. From this we can understand that the state which is recorded is indeed a consistent one.

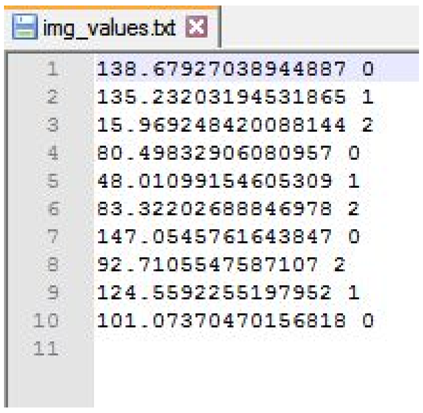


```
Entered print Global State function
record 12falserecord 13false
Entered record_12==false && record_13==false
Came out of  record_21==false && record_23==false Line:530
Came out of  record_31==false && record_32==false  Line:559
At Line:565
channelState_12 is0
  channelState_13 is51
 channelState_21 is0
  channelState_23 is0
 channelState_31 is0
  channelState_32 is0process 1's balance is1068process 2's balance is868process 3's balance is1013
```

Fig. 3: Global Snapshot output printed on Master/Initiator Node

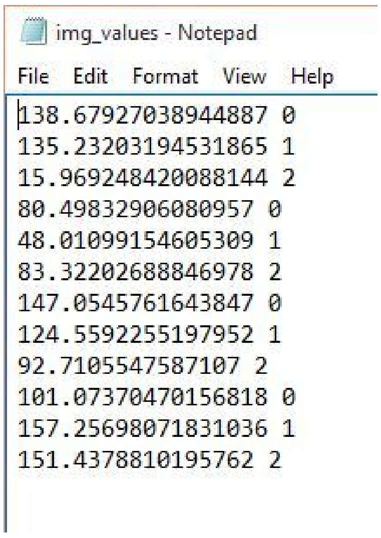
2.
We then implemented Image Processing Application of Calculating Mean Intensity of an Image and storing it in the node's storage in form of text files and also in other nodes.
Thus, every node has the same image intensity values with information on which node generates it.

To evaluate the consistency, we stopped the system abruptly on all the nodes (till then, one checkpoint had been achieved by Master/Initiator Node), and then observed the values at each node to check whether global consistency is achieved.



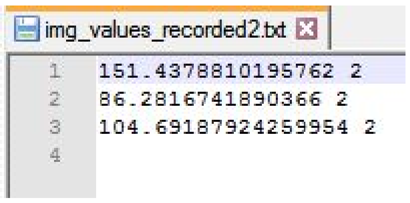
```
138.67927038944887 0
135.23203194531865 1
15.969248420088144 2
80.49832906080957 0
48.01099154605309 1
83.32202688846978 2
147.0545761643847 0
92.7105547587107 2
124.5592255197952 1
101.07370470156818 0
```

Fig. 4: Process state at Initiator



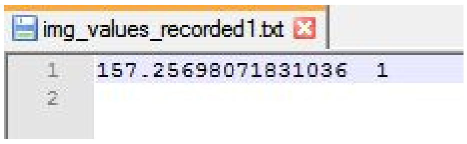
```
138.67927038944887 0
135.23203194531865 1
15.969248420088144 2
80.49832906080957 0
48.01099154605309 1
83.32202688846978 2
147.0545761643847 0
124.5592255197952 1
92.7105547587107 2
101.07370470156818 0
157.25698071831036 1
151.4378810195762 2
```

Fig. 5: Node 1's process state



```
151.4378810195762 2
86.2816741890366 2
104.69187924259954 2
```

Fig. 6: Initiator's incoming channel 2 state



```
157.25698071831036  1
```

Fig. 7: Initiator's incoming channel 1 state

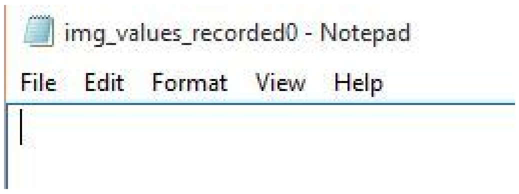
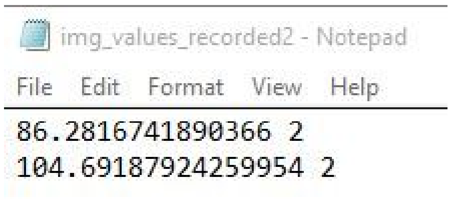


Fig. 8: Node 1's incoming channel 1 state (it is recorded as empty at the time it received a marker)



```
86.2816741890366 2
104.69187924259954 2
```

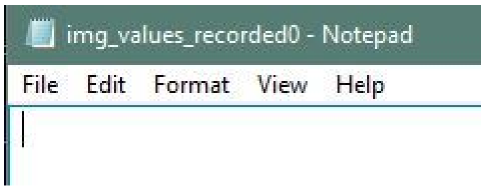
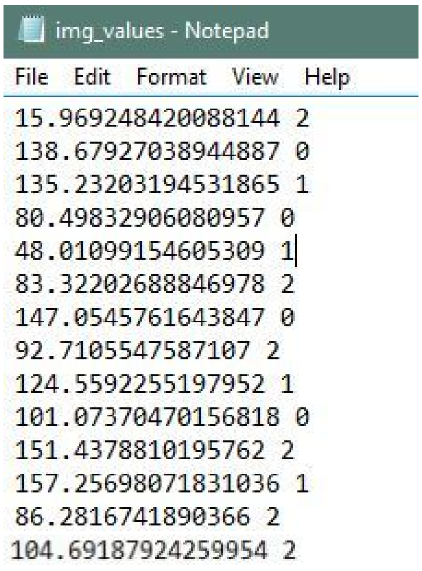


Fig. 9: Node 1's incoming channel 2 state

Fig. 10: Node 2's incoming channel 1 state (recorded as empty)



```
15.969248420088144 2
138.67927038944887 0
135.23203194531865 1
80.49832906080957 0
48.01099154605309 1
83.32202688846978 2
147.0545761643847 0
92.7105547587107 2
124.5592255197952 1
101.07370470156818 0
151.4378810195762 2
157.25698071831036 1
86.2816741890366 2
104.69187924259954 2
```

Fig. 11: Node 2's process state

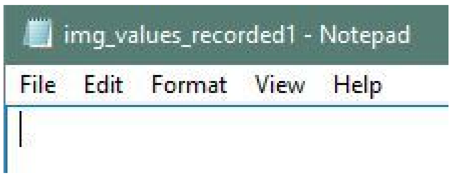
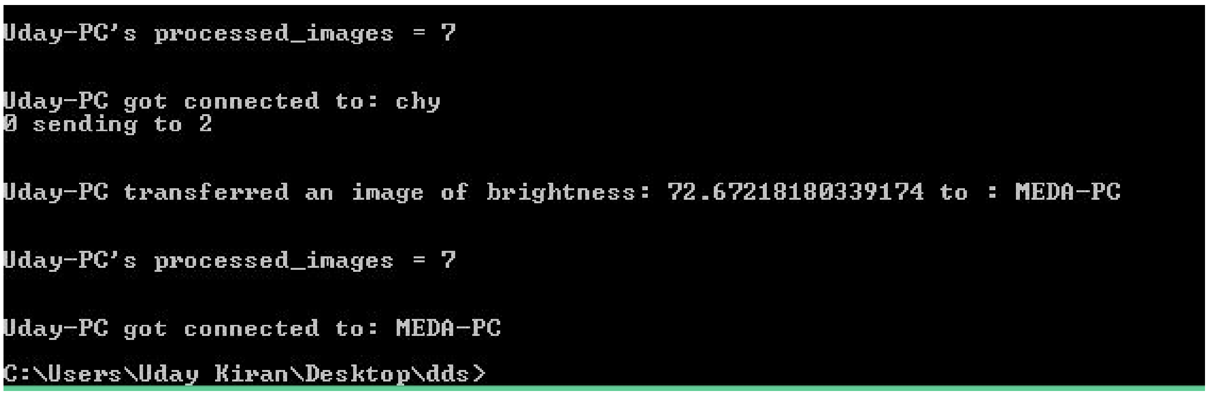


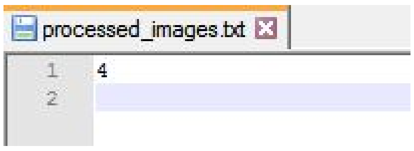
Fig. 12: Node 2's incoming channel 2 state (also recorded as empty)

After the checkpointing is done, from img_values txt file, we observe that node's state. This file has the intensity value of image stored in it, the second column in this file shows which node had generated this value. The incoming channel states of each node is observed in the two "img_values_recorded<other_nodeIDs>.txt" files. After the checkpoint if we get all the values in each of node's image_values.txt file and the two recorded files img_values_recorded<other_nodeIDs>.txt together, we find that all the three nodes have same image intensity values albeit in the order the respective node processes and receives. Once we restart the algorithm after artificially crashing (ctrl+c on each node to kill the image processing midway) all the three nodes the process continues from the last checkpoint. We observed that the state obtained at the end of processing all the images without artificially crashing the nodes is same as the end state obtained after artificially crashing all the three nodes and restarting. From this we can understand that the global snapshot being recorded by our algorithm is indeed consistent. Using our approach **Replication** is achieved at every node and if a Node crashes, we still have Image Intensity values of that specific node in other two nodes' database. We recall that each node works on only its share of 100 images. From this we can say that our algorithm achieves **Parallelism**, **Tolerance** to crashes and most importantly takes **Checkpoints** at periodic intervals.



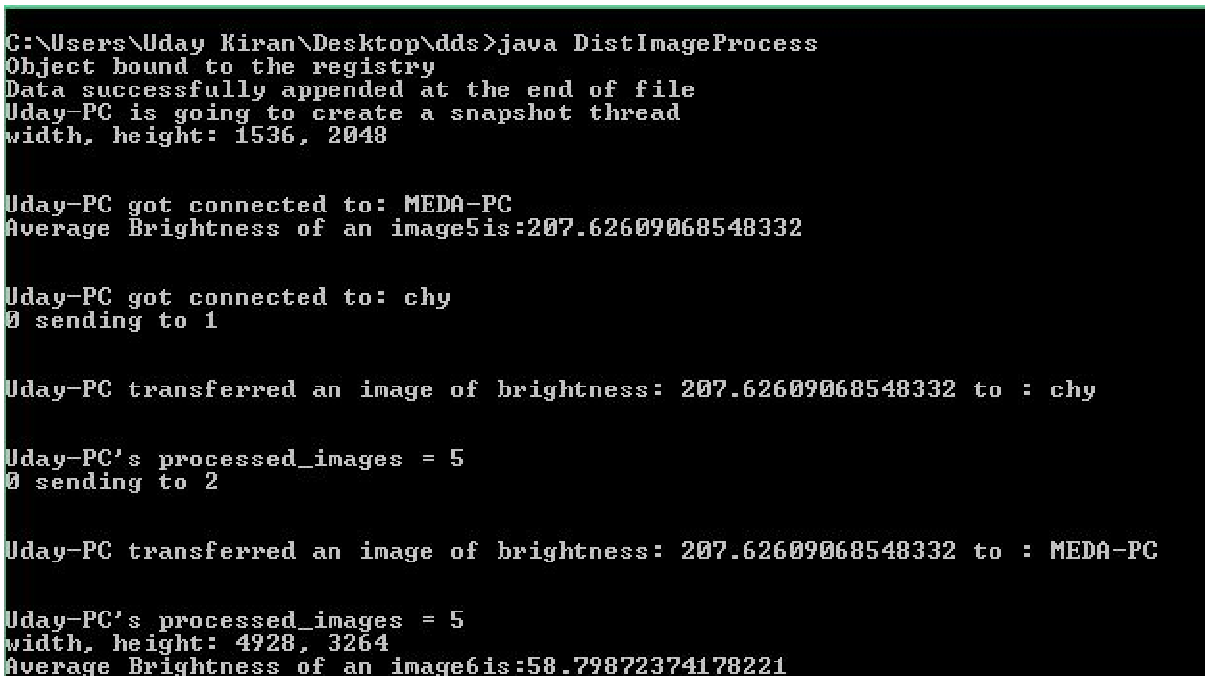
```
Uday-PC's processed_images = 7


Uday-PC got connected to: chy
0 sending to 2


Uday-PC transferred an image of brightness: 72.67218180339174 to : MEDA-PC


Uday-PC's processed_images = 7


Uday-PC got connected to: MEDA-PC

C:\Users\Uday Kiran\Desktop\dds>
```

Fig. 13: The number of images the initiator had already processed when the system crashes



```
processed_images.txt
1   4
2
```

Fig. 14: The number of processed images at the time of the last checkpoint in the initiator



```
C:\Users\Uday Kiran\Desktop\dds>java DistImageProcess
Object bound to the registry
Data successfully appended at the end of file
Uday-PC is going to create a snapshot thread
width, height: 1536, 2048


Uday-PC got connected to: MEDA-PC
Average Brightness of an image5is:207.62609068548332


Uday-PC got connected to: chy
0 sending to 1


Uday-PC transferred an image of brightness: 207.62609068548332 to : chy


Uday-PC's processed_images = 5
0 sending to 2


Uday-PC transferred an image of brightness: 207.62609068548332 to : MEDA-PC


Uday-PC's processed_images = 5
width, height: 4928, 3264
Average Brightness of an image6is:58.79872374178221
```

Fig. 15: The initiator restarts from the last checkpoint, soon after system restart from any arbitrary failure (<images processed until checkpoint>+1)

V. WORK BREAKDOWN

The major part of the project was implementing the core of the Chandy Lamport algorithm. The other aspects were: achieving system dependability (replication), developing the image processing code on top of our algorithm, printing relevant messages for debugging and understanding of process flow. We have developed the entire code ourselves. We referred to the research paper discussed in class for the algorithm; we have utilized online resources for learning to use the RMI API in Java. Our team's work split was done as follows:

• Chandy Lamport algorithm implementation • Data replication and dependability measurements • Restart from checkpoint feature	Prapurna Jayakrishna, Uday, Mrunmayi
• Image processing application • Creation of image database • Using remote communication API	Ankul, Rudra

VI. CONCLUSION

- Chandy Lamport Algorithm solves the problem of checkpointing of various nodes
- Data Replication and Consistency is achieved using our approach
- This project demonstrates that various Image Processing Applications can be implemented with fault tolerance and reliability
- Our ultimate goal of implementing Chandy-Lamport Algorithm was achieved and we developed a Image Processing Application to demonstrate its functionality

VII. FUTURE WORK

- This work can be extended to be used in advanced Image Processing to calculate the Manhattan Distance between two images and store and communicate the difference of these images in various nodes in the System
- This parallelism of this project can be explored further to compute convolution and filtering operations.

VIII. REFERENCES

[1] K. Mani Chandy, Leslie Lamport - Distributed Snapshots: Determining Global States of Distributed Systems - http://users.ece.gatech.edu/~dblough/6102/papers/chandy.pdf
[2] Java Tutorial on RMI Applications:
https://docs.oracle.com/javase/tutorial/rmi/overview.html
[3] Message Passing Interface (MPI) Tutorial: https://computing.llnl.gov/tutorials/mpi/
[4] Code snippets for getting RGB values for a pixel :
http://docs.oracle.com/javase/7/docs/api/java/awt/image/BufferedImage.html#getRGB(int,%20int)
[5] Image Database: https://snippets.khromov.se/stock-photo-archive-zip-77-images/