Indiana University-Purdue University Indianapolis

Department of Computer and Information Science

CSCI 53700

Fall 2017

Assignment - 1

AKHIL NAYABU
ID: 2000075395

Table of Contents

I. Introduction

This assignment is based on clock consistency, associated drifts, inter-process communication. In this assignment, we implemented Lamport's logical clock and achieved clock consistency using Berkley Algorithm. Here in this assignment, we were asked to implement a distributed system where we have defined four process objects which have the Lamport logical clock implemented along with a Master Object which has its own logical clock implemented. Our goal was to achieve consistency among the logical clocks using Berkeley Algorithm. As part of this assignment we were asked to provide analysis on the clock drifts and synchronization when this distributed system was run for a long time. Also, we did analyze the behavior of the system in terms of clock drifts and synchronization when we have arbitrary or byzantine failures in any given Process Object. Apart from implementing a simple distributed system, we have implemented a simple encryption algorithm which is used to exchange logical clock values between the master object and the process objects.

II. Design – Summary:

A. Main Design Choices:

- Implemented a simple client-server system using server-socket programing of java.
- Master Object is selected as server since it needs to communicate with all the process objects, and since process objects need not communicate with each other they can be considered as clients.
- On each client connection, a new thread is created or every new incoming connection is made on a new thread.
- Since send, receive, and internal computation occur randomly based on their probability, I have initially generated random numbers between 0-2 and decision of which event is to be occurred is based on random number which was generated.
  0 → Internal Computation (In my system it's just an increment in logical clock value)
  1 → send event (where process object sends its clock value to master object)
  2 → receive event (where client will receive offset and adjust its clock value based on returned offset.
  So, the probability for each event was 33.33%.
- On each event, logical clock value is incremented by 1.
- Current clock value is only sent to master object when client has hit send event i.e. every time client has a send event, it will increment the clock value and send the current clock value to master object.
- Master Object increases its counter on every internal event.
- Master Object calculates the offset every time when it receives a clock value from process object, once the offset is calculated it sends back the offset to the client for correction.
- On every receive event from the client Master object will return the offset value which was placed in the queue after the offset was calculated last time.

B. Other Design Choices:

- To identify different clients on master object, I have passed client id as 1 st argument on command line. Having client id helps master to identify which object has sent its clock value or to also predict which process object has sent a receive request.
- To make one of the process object faulty, I have passed a flag (y/n) as 2 nd argument on command line. So, if the 2 nd argument is y then that process object exhibits the byzantine failure. In my failure, I have just incremented the clock value by +100. Error % for byzantine was 20.
- For encryption, every time before client has sent the clock value, it is being encrypted with key. In my code, I have just added +10 before the clock have is sent and once clock value is received on master -10 is performed on received clock value.

III. Detailed Design

As part of this distributed system we have defined four process objects and one master object. In my system, I have represented the Master Object by MasterThread.java and the Process Object is represented by Client.java. Also, we have defined MasterServer.java to facilitate concurrent communication between Master Object and Process Objects.

A. MasterThread.java (Master Object):

In my implementation, I have defined the Master Object to behave like a Server where this program would always be listening for messages from the process objects. This Master Object has the implementation of Server Socket. To allow multiple process objects to communicate concurrently with the Master Object, we have made the Master Object i.e. MasterThread class to support multi-threading. Hence we have extended this master object class with Java Thread class. Various methods that are defined in my Master Object are:

1. public void run()
   Once we have a socket connection established between the master and the process object, here we receive the incoming messages from Process objects and occasionally increment the logical clock of the master by 1.

2. synchronized void receive()
   - This method in Master Object is used to receive a message from the Process objects.
   - This message can be requesting the master to send the latest clock offsets or sending the master with a process latest logical clock value.
   - Message Format: The message from the PO has three segments separated by a delimiter @.
     <PROCESS_ID>@<EVENT>@<LOGICAL_CLOCK>
     Example: 1@SEND@10
     2@RECEIVE@

     PROCESS_ID: This indicates which specific Process Object is sending this message.

EVENT: Event indicates whether the Process Object is:

SEND: Sending the master with latest logical clock information. In such an event the logical clock value is always populated in the last segment of the message.

RECEIVE: The PO is requesting the MO to send the latest logical clock offsets so that the PO can corrects logical clock.

LOGICAL_CLOCK: Only in case of SEND event we receive the logical clock value is sent by the PO. This logical clock value is sent in an encrypted manner. We need to decrypt this logical clock value before using it.

- We read the message from the socket.
- We tokenize the received message from the PO into three tokens: PROCESS_ID, EVENT, LOGICAL_CLOCK
- If the event received is SEND,
  - Decrypt the logical clock value which was sent by the PO using the methods in EncryptionHelper.java
  - Update the logical clock value of this specific PO on the Master Object.
  - Calculate the new average of the logical clocks of MO and four PO's.
  - Set the MO's logical clock value to the average.
  - Calculate the offset for each of the PO's.

```
poOffSet1 = average - poClock1;
poOffSet2 = average - poClock2;
poOffSet3 = average - poClock3;
poOffSet4 = average - poClock4;
```

  - Send the new offset value to the PO so that the PO can perform the logical clock correction on its end.
- If the event received is RECEIVE,
  - Based on the process ID received in the message, we send the new offset value from the MO to the PO by invoking the send().
  - Once we have sent the offset value to the PO, we mark the offset value of this requested PO to zero.
- This method is marked as synchronized to make sure that no two PO's execute this in parallel.

3. synchronized void send()
   - In this method, we write the new offset value into the socket.

4. private void randomCall()
   - This method will ensure that the Master Object's logical clock is incremented periodically. The logical clock is incremented by 1with a probability of 0.33.

5. synchronized void clockIncrement()

- We have defined the Master Object's logical clock to be AtomicInteger because this variable is shared between multiple threads that are created per PO. AtomicInteger ensures this logical clock is thread safe.
- This method increments this logical clock value by 1 in the Master Object.

6. Variables defined:

```
static final int TOTAL_NO_PO = 4;
private final AtomicInteger masterClockCounter;
private static int poClock1 = 0;
private static int poClock2 = 0;
private static int poClock3 = 0;
private static int poClock4 = 0;
private static int poOffSet1 = 0;
private static int poOffSet2 = 0;
private static int poOffSet3 = 0;
private static int poOffSet4 = 0;
```

B. MasterServer.java:

This class creates a server socket and creates a thread for the Master Object for every incoming socket connection from the Process Object.

C. Client.java: (Process Object)

In my implementation, Client.java represents the Process Object. In the process object we perform one of three events at a probability of 0.33. Three possible events are:

- Random Event: In case of this event we just do a logical clock Increment by 1.
- Send: In a send event, the PO send the current logical clock information to the Master so that the Master Object try to adjust the logical clocks accordingly if any clock drift is present.
- Receive: In a receive event, the PO requests the master to send the Latest logical offset. The master upon receiving the request, it sends back the PO with the new logical clock offset. Upon receiving this offset, the PO performs its logical clock correction.
- In case of a send or receive event also we perform a logical clock increment by 1.
- If we identify a certain PO is faulty, with a probability of 0.2, the PO's logical clock would be incremented by 100 instead of incrementing it by 1.

Various Methods in this class are:

1. static void main(String[] args)
   - In this method, we try to perform three main events based on certain probability. Here we either increment the PO logical clock, send message to MO or receive message from MO.
   - We accept the PO Process ID as the first command line argument for the PO. Also, we accept the faulty flag which is either a Y or N as the second command

line argument. This faulty flag is used indicate whether the PO has any arbitrary or byzantine failures.
- One of these above listed events are performed at a probability of 0.33
- In this method, we constantly keep performing these events; In case of send or receive events we perform socket communication with the Master Object.

2. private void receive()
   - This method is invoked in case of a receive event on PO.
   - In this method, we send a request to the Master requesting it to send the latest offset for this PO. This request is sent by writing it in the socket.
   - Once the MO sends the latest offset value, the new offset value is read from the socket.
   - Once the new offset value is read and decrypted, we correct the logical clock value and increment the logical by 1.
     clockCounter = clockCounter + receivedOffset + 1;
   - The received offset value can be positive or negative.

3. private void send()
   - This method is invoked in case of a send event on PO.
   - In this method, we send the latest encrypted logical clock value to the MO by writing this message into the socket.
   - Once the new MO receives this request and it calculates the new offset value of all the PO's and sends back the new offset value of the requested PO.
   - Once the PO receives back the new offset value, the PO performs the logical clock value correction and increments its value by 1.
   - The new offset value is obtained from the MO by socket communication.
4. private void clockIncrement()
   - This method accepts error indicator as an input. This flag is passed as Y in case of a faulty PO.
   - If the PO is not faulty the logical clock value is incremented by 1.
   - In case of an faulty PO, the logical clock value is incremented by 100.

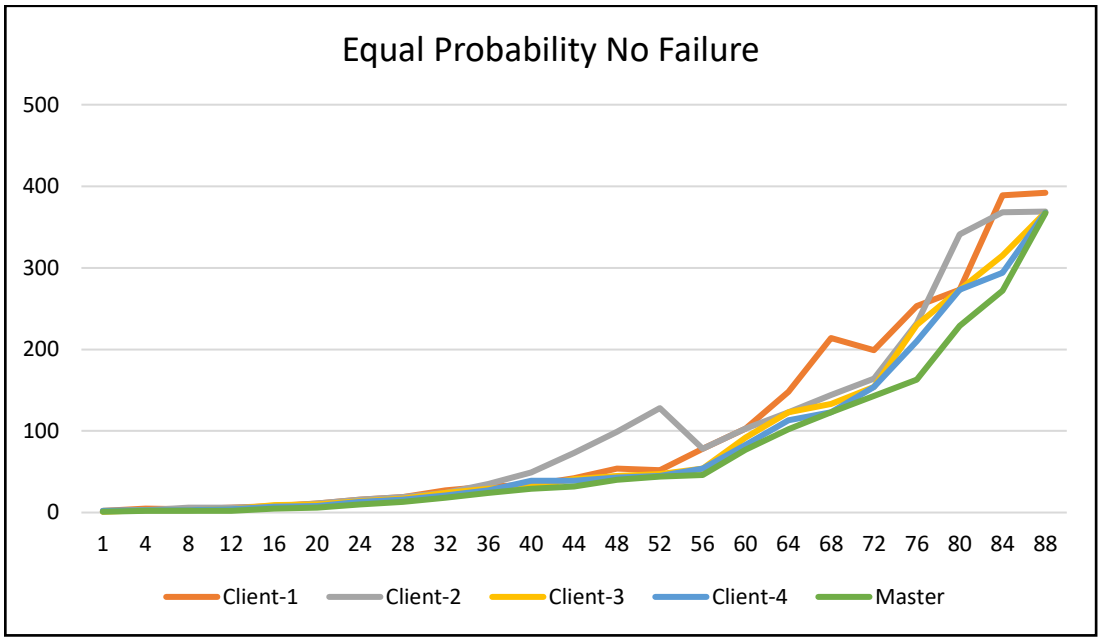
5. private int randomCall()
   - This method randomly generates value between 0 and 2.

D. EncryptionHelper.java

This is an utility class that I have written which will help performing encryption and decryption of messages.

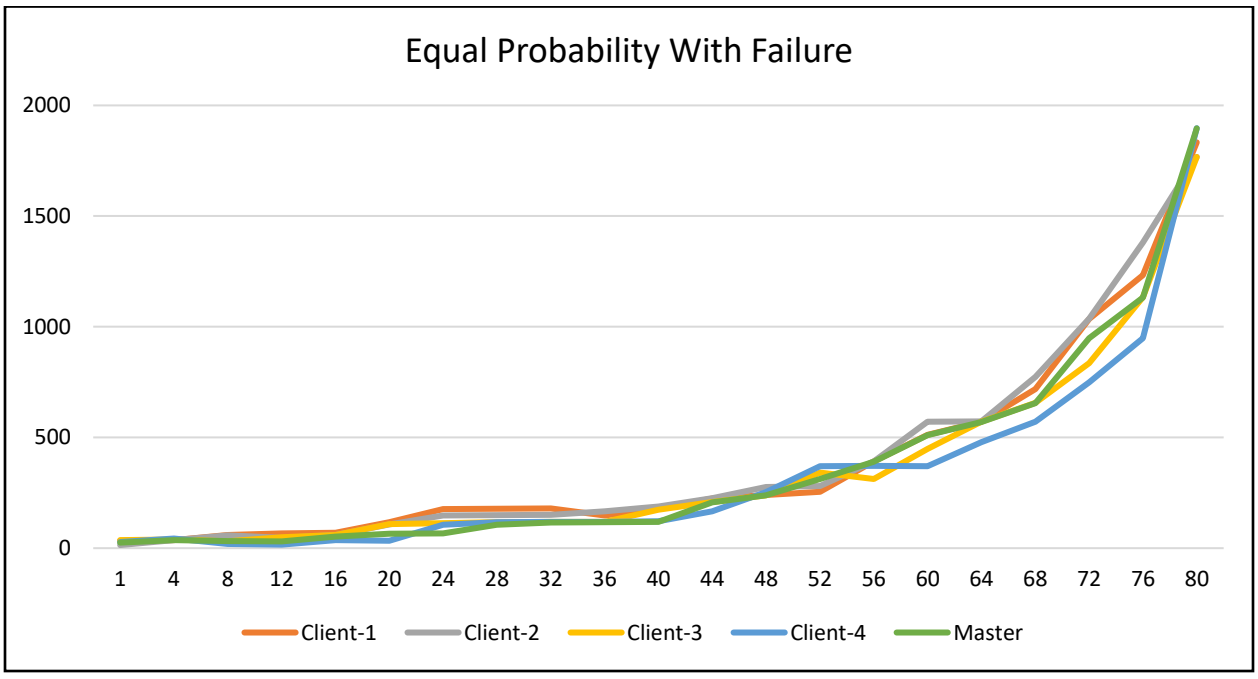
IV. Analysis & Discussion

A. System without failures and equal probability for all events



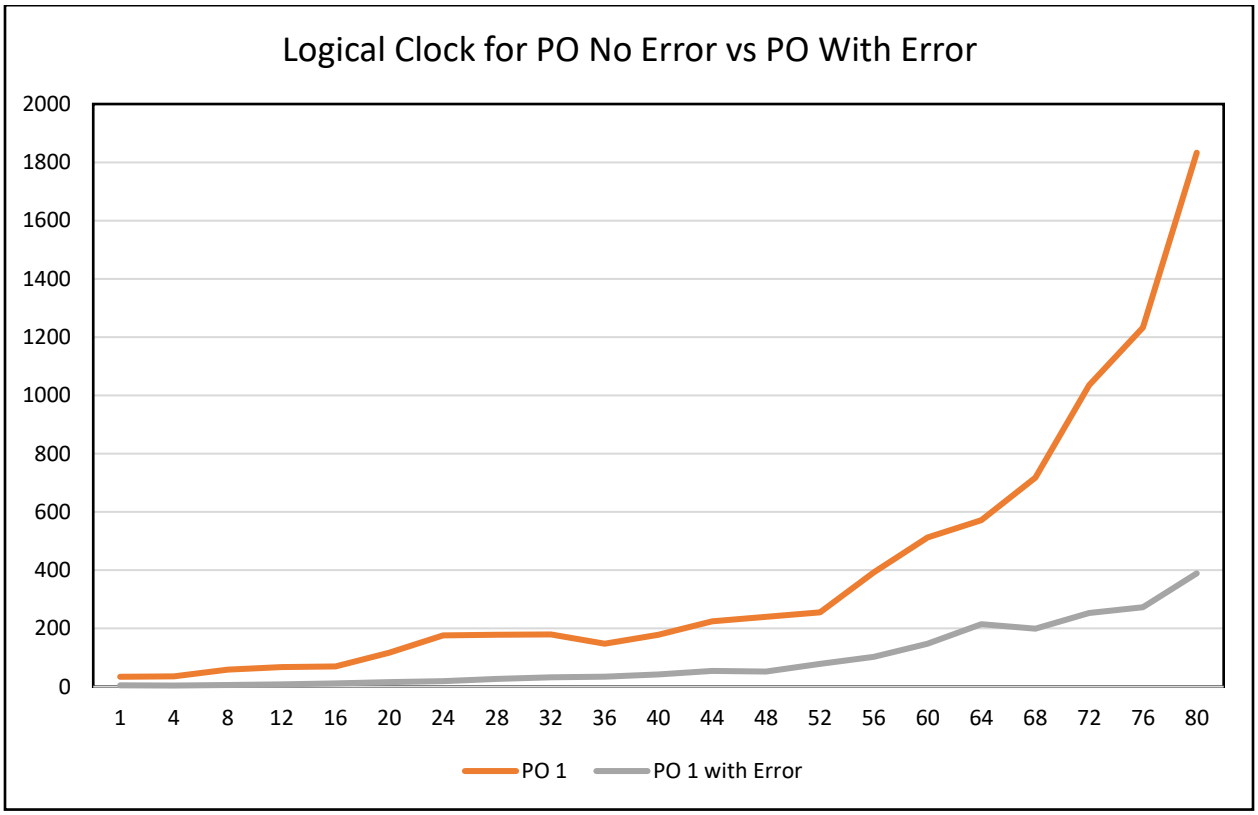
In the above graph, X-Axis depicts the time elapsed in seconds and Y-Axis represents the logical time of the Process Object / Master Object. The clocks were in sync initially and there was a clock drift that started to happen, but soon after the clock drifts got corrected.

B. System with 20% byzantine failure and equal probability for all events



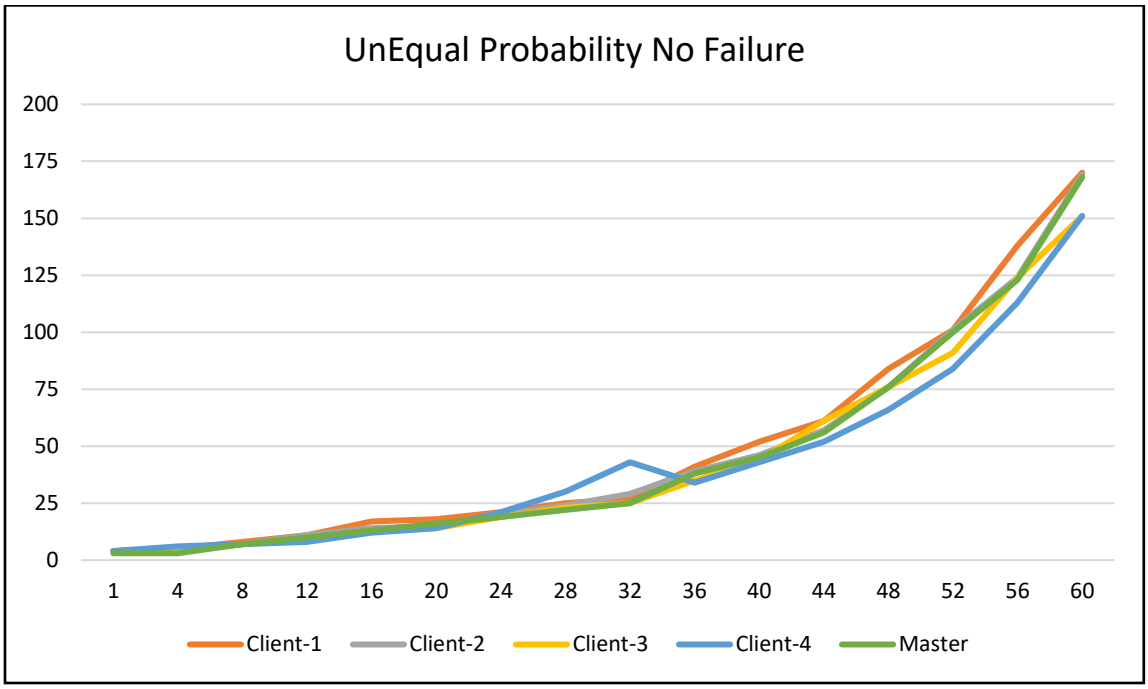
In the above graph depicts the logical clocks graphical representation when the distributed system had one PO with 20% byzantine failure. X-Axis represents time elapsed in seconds and Y-Axis represents the logical clock. As we notice that in case of the failure PO, we see that the logical clock values went high when compared to that of the no failures distributed system. Even though the logical clock values were too high, yet the overall logical clock values of the PO and MO were almost in sync.

C. Comparison of logical clock values for a Process Object No Error vs Error



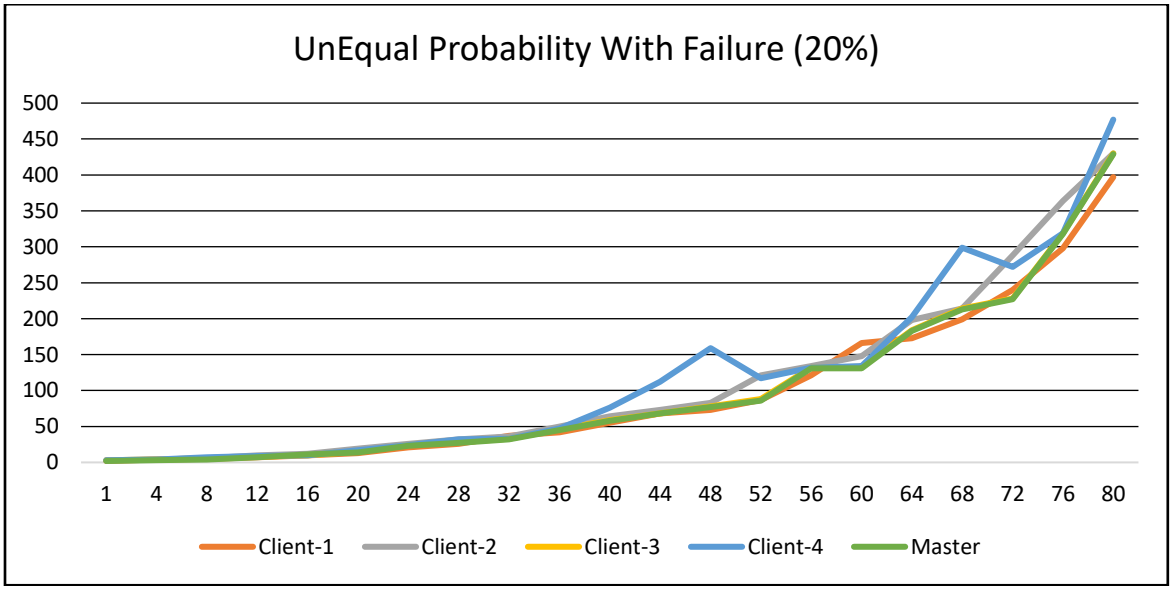
From the above graph, X-Axis represents the time elapsed in seconds and Y-Axis represents the logical clock values. From this graph, we notice the different in the logical clock value for a given PO for scenarios when NO FAULT vs FAULTY PO.

D. System without failures and NOT equal probability for all events



From the above graph, X-Axis represents time elapsed in seconds and Y-Axis represents the logical clock values. In this graph depicts the reading from the system where we have un equal probability for the events that are generated in this system. This is NO fault system. Here we see that irrespective of the probability we see that logical clock PO's and MO are always kept in sync or in close by correcting range.

E. System with 20% byzantine failure and NOT equal probability for all events



From the above graph, X-Axis represents time elapsed in seconds and Y-Axis represents logical clock values. This graph depicts readings from the system where we have unequal probability for the events that are generated and we have a faulty PO in the system. As we notice from the graph, we see that one PO which is faulty has the logical clock value abruptly increased by 100, but eventually in due time, all the PO's and MO are brought into sync.

Summary:
After going through all the above observations one could say that no matter how far ahead or far behind the clocks were, they were eventually pulled backwards or forwards based on the calculated offset which brought clocks closer to each other.
And when byzantine error occurred all the clocks were brought into that faulty clock state but still maintaining clock consistency as clocks were still closer to each other.
Given the nature of the system i.e. random probability for each event, clocks of each object will never be same, but they will be closer to each other with certain ±α.

V. Interaction and Failure Models

Interaction Model:

An interaction model states that a distributed system consists of many interacting processes. It will difficult to predict the process proceeds and timings of the transmission of messages.

Variations in Interaction Model:
Synchronous distributed system: time to execute each process will have bound. Timeouts are used to detect failures.
Asynchronous distributed system: no assumptions made on time take to execute each process. Event ordering cannot be dependent on time.

Failure Models:

Omission Failure: these are the failures which include process being halted or will stop executing father. Can be detected by invocation messages. Fail-Stop process crash, if other process can detect certainly that the process has crashed. Communication omission failure is a failure when communication channel does not transport a message from a sender's outgoing buffer to receiver's incoming buffer.

Arbitrary Failure: Byzantine failures, there are the worst possible failures semantics like sending incorrect values, omits intended processing and corruption of message. This failure was implemented in the assignment and was discussed earlier.

VI. Execution/Sample Run

Steps on how to compile and run are included in the readme file. The following is a screenshot from of the sample runs.