Programming Assignment 2: Chandy-Lamport Snapshot for a PassToken Application

Due Monday Oct. 7, 9 pm

1 Overview

The Chandy-Lamport Snapshot Algorithm [2] is a distributed algorithm for capturing a consistent state of an asynchronous system. It is useful for debugging and checkpointing. The purpose of this lab is to see how you will use Chandy-Lamport in practice for an application.

2 Lab Task

You will first implement an application `PassToken` where processes self-organize in a ring and pass a special message `Token` along the ring. Then, one or several processes will start the *Chandy-Lamport Snapshot* algorithm that will get all processes to checkpoint their state. You can assume the system has five processes. The ring organization is defined in a hostsfile that all processes know. More details below. To receive full credit, you will need to pass all test cases described below. All messages must be printed in the exact formats specified below.

2.1 PART 1: `PassToken` Application (30 points)

In this part, you are going to create a distributed application for which snapshotting will be useful. Each process maintains a counter called STATE. While the processes are organized in a ring for the purpose of passing the token, each process should be connected to every other process using TCP (this will be important in the next section).

Ring formation. All processes read a hostsfile that allows them to self-organize in a ring. Each process needs to know its successor and predecessor in the ring and maintain channels (TCP connections) with all the other processes. The processes are organized in a ring sorted by ID (with the highest ID wrapping around to connect to the lowest). In the hostsfile, the ID is the row number starting at one. Note that all the information about the ring should be inferred from the hostsfile, there are no messages sent between processes at this point.

When a process starts up, it should print its process ID and initial state in the form:

```
{proc_id: ID, state: STATE, predecessor: preID, successor: sucID}
```

Above `preID` and `sucID` are the IDs of the predecessor and successor in the ring for the process printing the info.

Token processing. A token is a special message identified by its type. There is only ever one token in the system at a time, and a process passes the token by sending a `Token` message to the process that follows it in the logical ring formed by the five processes. The process that will first have the token will be passed with the the "-x" CLI flag when starting the system.

To pass the token, a process sends the `Token` message to its successor in the ring. Every time a process receives a token, it increments its local counter STATE, prints the new current value of

STATE, sleeps for t seconds, then sends the token to its successor. The application should not terminate. The state should be printed as:

```
{proc_id: ID, state: STATE}
```

Every time the token is passed, the sender and receiver should print

```
{proc_id: ID, sender: SENDER_ID, receiver: RECEIVER_ID, message:"token"}
```

TESTCASE 1 (30/30 points): The n^{th} process in the hostsfile should start with the token, and the token should be passed in a full circle around the ring, $1 \leq n \leq 5$. Full points for one complete revolution around the ring, though it does not need to stop after one revolution.

2.2 PART 2: Chandy-Lamport Snapshot (60 points)

In this part, you will integrate snapshotting functionality into `PassToken`. The snapshot algorithm you will be using is Chandy-Lamport.

2.2.1 Algorithm

This algorithm involves recording the state of each process as well as any messages that were in-flight and were not captured in any process's state. The algorithm maintains two types of data. First, for every process $p \in P$, it records a state S_p. For this lab, the state will be a monotonically increasing integer that always starts at zero. The second type of data is the *channel*. A channel is a FIFO queue. Each process maintains an incoming and outgoing channel for each process it is connected to. We write a channel from p_x to p_y as C_{xy}. For example, if p_1 is connected to p_2 and p_3, then it will maintain four channels, C_{12}, C_{13}, C_{21}, and C_{31}.

The algorithm uses a single type of message called `Marker`. A process will follow one of two possible procedures when it receives a marker. Which procedure it follows depends on whether or not it has already received a marker from another process.

If the process p_i receives a marker from process p_k and *has not* already received a marker, the process will

1. record the state S_{p_i}
2. mark as "closed" the channel that the marker was received on (C_{ki})
3. send out markers on all its outgoing channels
4. start recording messages received on all incoming channels (except C_{ik})

If the process p_i receives a marker from process p_k and *has* already received a marker, the process will stop recording on channel C_{ik}. In other words, it will close the channel. The `Marker` should not be included in the recorded queue.

The snapshot begins at a single process where the process acts as though it has received a marker, i.e., it starts at step two. The first process can skip step two because it will not have received the marker on a channel. The snapshot ends when all processes have closed their incoming channels. The final snapshot thus contains the states of each process and the FIFO incoming channels stored at each process.

2.2.2 Implementation

In this part, you will implement the above algorithm. First, you will draw a state diagram [1] for a single process running Chandy-Lamport. Computer scientists often use state diagrams to present a high-level overview of the behavior of a system. **Make sure you include the state diagram in your report**. You are advised to have the state diagram ready before you start coding. The diagram will come in handy during implementation. This algorithm should be part of the same program as the application from PART 1 (Section 2.1). An important focus will be on how the token is processed in relation to the `Marker`. The marker is just a special message, i.e. identified by its type.

Snapshotting is part of the functionality of `PassToken`, so the `Marker` should be sent through the same TCP connection as the token (not a separate dedicated connection). You can use either a duplex (two-way) TCP connection per pair of processes, or have two unidirectional TCP connections per pair. The latter is recommended as it will be easier to implement.

The program should accept a CLI parameter m specifying a delay. This delay tells the process how long it should wait (in seconds) between when it receives the `Marker` to when it sends out its own markers to the rest of the processes. (This is to try to create more interesting scenarios.) The program should also accept a CLI parameter s indicating how long to delay before initiating the snapshot, if a process receives this flag it should wait until its state is equal to s before initiating a snapshot.

Every time a marker is sent, print

```
{proc_id:ID, snapshot_id: SNAP_ID, sender:S_ID, receiver:R_ID, msg:"marker",
state:STATE, has_token:YES/NO}
```

The key `snapshot_id` is a unique identifier for the snapshot as provided through the command line interface. It can be any 32-bit integer as long as no other snapshot has the same ID with high probability.

When the snapshot begins executing, print

```
{proc_id:ID, snapshot_id: SNAP_ID, snapshot:"started"}
```

When the channel is closed, print

```
{proc_id:ID, snapshot_id: SNAP_ID, snapshot:"channel closed", channel:S_ID-R_ID,
queue:[CHANNEL_VALUES]}
```

CHANNEL_VALUES is a **comma-separated** list of all the messages in the closed queue.

When the process has finished the snapshot (i.e., all the incoming channels have been closed), print

```
{proc_id:ID, snapshot_id: SNAP_ID, snapshot:"complete"}
```

TESTCASE 2 (20/60 points): Perform a snapshot from a single process which starts after a short delay ("-s 2"), with token propagated every 0.2 seconds ("-t 0.2"). Use a marker delay of 2 seconds ("-m 2"). Full points if all processes finish the algorithm (checkpoint the state and received markers on all incoming channels).

TESTCASE 3 (20/60 points): Perform a snapshot from a single process which starts after a short delay ("-s 2"), with tokens propagated more after a 0.2 second delay ("-t 0.2"). Use a marker delay of 2 seconds ("-m 2"). Full points if the snapshot is correct including all the states and queues. Partial credit is available for getting some of the snapshot states and queues correct.

TESTCASE 4 (20/60 points): Perform a snapshot from a single process which starts after a short delay ("-s 2"), with tokens propagated more after a 0.2 second delay ("-t 0.2"). Use a marker delay of 2 seconds ("-m 2"). When that snapshot completes, start a new snapshot from a different process. This can be achieved using a longer snapshot delay (e.g., "-s 30"). Full points if both snapshots are correct including all the states and queues. Partial credit is available for getting some of the snapshot states and queues correct.

TESTCASE 5 (10 points extra credit): Perform a snapshot from a single process which starts after a short delay ("-s 2"), with tokens propagated more after a 0.2 second delay ("-t 0.2"). Use a marker delay of 2 seconds ("-m 2"). *Before* that snapshot completes, start a new snapshot from a different process (e.g., "-s 2"). Full points if both concurrent snapshots are correct including all the states and queues.

2.3 Implementation details

You need to implement this program in `C/C++`, `Java`, or `Go`. You will be using Docker and Docker Compose to package your program. Instructions on how to use Docker and Docker Compose for this project can be found in the Docker tutorial. You will be provided with a `docker-compose.yml` file for each testcase. Your program should be able to use that file without modification.

***Note*:** You **MUST** write code that **compiles and runs** in your container.
***Note*:** All messages must be printed to `stderr` and all printed messages must be terminated with a newline (`\n`).

Building and running instructions below. Note that you will need to build the Docker container before orchestrating with Docker Compose:

```
Building:
    docker build . -t prj2

    Running this in your project's directory should
    build your project's Docker image using your
    Dockerfile. It should copy over the code and
    hostsfile to the image and compile your project.

Single container usage:
    docker run --name <hostname> --network <network> --hostname <hostname> prj2 -h <hostfile> \
               -t <token_delay> -m <marker_delay> [-s <snapshot_delay> -p <snapshot_id>] [-x]

  --name <hostname>
    This specifies the name of the Docker container,
    this name should match the one in your hostfile

  --network <network>
    This is the user-defined Docker network your project
```

```
    will run in, refer to tutorial in additional instructions
    to learn how to create one and why you need one.

  --hostname <hostname>
    This specifies the hostname of the Docker container,
    this name should match the one in your hostfile. It can be used by other
    containers in the same bridge network (e.g., like the one created in the
    tutorial, or the one used in the Docker Compose template).

  -h <hostfile>
    The hostfile is the path to a file inside your Docker container
    that contains the list of hostnames that the Docker container is
    running as (specified using the --name flag above). It assumes
    that each container is running only one instance of the program.
    It should be in the following format.

    container1
    container2
    ...

    All the processes will listen on the same port.
    The line number indicates the identifier of the process
    which starts at 1.

  -x
    The process that starts with the token, it's starting state should be one
    and all other processes should have a state of zero.

  -t <token_delay>
    A float representing how long the process should sleep between
    receiving the token and passing the token to its successor.

  -m <marker_delay>
    A float representing how long the process should sleep between
    receiving the marker and sending the marker to the other processes.

  -s <state>
    An integer indicating that this process should initiate the snapshot
    once its state has reached <state>.

  -p <snapshot_id>
    An integer representing the unique identifier of the snapshot. Can only
    be provided to the process that is starting the snapshot (i.e. the process
    that recieves ``-s <state>'').

Orchestration:
   docker compose -f [PATH TO COMPOSE FILE] up
```

3 Grading

The project is manually graded by your TA and not by autograder. Total: 100 points.

- Implementing the token passing application and passing the test case: 30 points

- Implementing the Chandy-Lamport algorithm and passing the test cases: 60 points
- Extra Credit: Implementing concurrent Chandy-Lamport snapshots and passing the test case: 10 points
- Code inspection: 10 points

4 Submission Instructions

Your submission must include the following files:

1. The **SOURCE** and **HEADER** files (no object files or binary)
2. A **Dockerfile** to build your project
3. A **hostsfile.txt** that we can use to test your project
4. A **README** file containing your name, any additional instructions to run your code and anything you would like us to know about your program (like errors, special conditions, etc.)
5. A **REPORT** describing your algorithm.

Submission is through Gradescope.

5 Additional resources

You may find the following resources helpful

- Socket programming: `https://beej.us/guide/bgnet/html/`
- Unix programming links: `https://cse.buffalo.edu/~milun/unix.programming.html`
- C/C++ programming link: `https://www.cplusplus.com/`
- `https://github.com/iowaguy/docker-tutorial`

References

[1] State diagram. `http://en.wikipedia.org/wiki/State_diagram`.

[2] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.