ARIZONA STATE UNIVERSITY

CSE 434, SLN 11057 — Computer Networks — Spring 2023

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday, 02/05/2023; Milestone due Sunday, 02/19/2023; Full project due Sunday, 03/12/2023

The purpose of this project is to implement your own peer-to-peer application program in which processes communicate using sockets to implement CHECKPOINT, a checkpointing and recovery technique for maintaining a consistent global state in a distributed banking application with failures.

- You may write your code in `C/C++`, in `Python`, or in `Java`; no other programming languages are permitted. Each of these languages has a socket programming library that you **must** use for communication.
- This project may be completed individually or in a group of size at most two.
- Each group **must** restrict its use of port numbers to prevent the possibility of application programs from interfering with each other. See §4.3 to determine the port numbers assigned to your group.
- You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub or similar. Your code repository **must** be *private* to prevent anyone from plagiarizing your work. It is expected that you will commit changes to your repository on a regular basis.

The rest of this project description is organized as follows. §1 begins with various definitions of state in a distributed system and provides algorithms to capture a consistent global state. Following an overview of the architecture of the CHECKPOINT application in §2, the requirements of the peer-to-peer protocol are provided in §3. Some issues for you to consider in your implementation are found in §4. The submission requirements for the milestone and full project are described in §5.

1 Global State in a Distributed System

The computers in a distributed system don't share memory or a common clock. As a result, an up-to-date state of the entire system is not easily available to any individual process. Such a global state may be useful for many reasons, such as reasoning about the system's behaviour, debugging, or recovering from failures. To understand the definition of a global state and what it means for it to be consistent we need to first define a local state [3].

1.1 Local State

For a site (computer) S_i, its *local state* at a given time is defined by the local context of the distributed application. Let LS_i denote the local state of S_i at any time. Let $send(m_{ij})$ denote the send event of a message m_{ij} by S_i to S_j, and $receive(m_{ij})$ denote the receive event of message m_{ij} by S_j. Let $time(x)$ denote the (local) "time" at which the state or message x was recorded.

For a message m_{ij} sent by S_i to S_j:

- $send(m_{ij}) \in LS_i$ if and only if $time(send(m_{ij})) < time(LS_i)$, according to S_i's clock.
- $receive(m_{ij}) \in LS_j$ if and only if $time(receive(m_{ij})) < time(LS_j)$, according to S_j's clock.

For local states LS_i and LS_j of any two sites S_i and S_j, two sets of messages are defined. These sets contain messages sent from computer S_i to computer S_j.

- **Messages in transit:** $transit(LS_i, LS_j) = \{m_{ij} | send(m_{ij}) \in LS_i \wedge receive(m_{ij}) \notin LS_j\}$.
- **Inconsistent messages:** $inconsistent(LS_i, LS_j) = \{m_{ij} | send(m_{ij}) \notin LS_i \wedge receive(m_{ij}) \in LS_j\}$.

1.2 Global State

A *global state* GS of a system is a collection of the local states of its sites, *i.e.*, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ where n is the number of sites in the system. A global state $GS = \{LS_1, LS_2, \ldots, LS_n\}$ is *consistent* if and only if

$$\forall i, \forall j : 1 \leq i, j \leq n, inconsistent(LS_i, LS_j) = \emptyset.$$

That is, in a consistent global state, for every message received there is a corresponding send event recorded in the global state. In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded. In Figure 1, the global state $\{LS_{12}, LS_{23}, LS_{33}\}$ and $\{LS_{11}, LS_{22}, LS_{32}\}$ correspond to consistent and inconsistent global states, respectively.

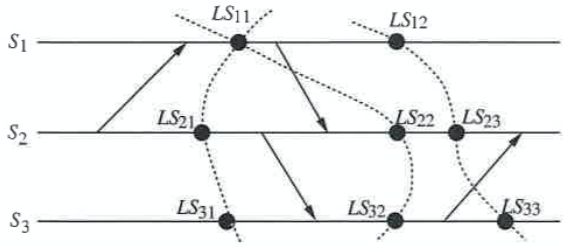


Figure 1: Global states in a distributed computation.

While the definitions given are for a group of sites, they can be applied to a group of cooperating processes by replacing sites with processes in the definitions. For example, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ represents a global state of n cooperating processes, where LS_i is the local state of process P_i.

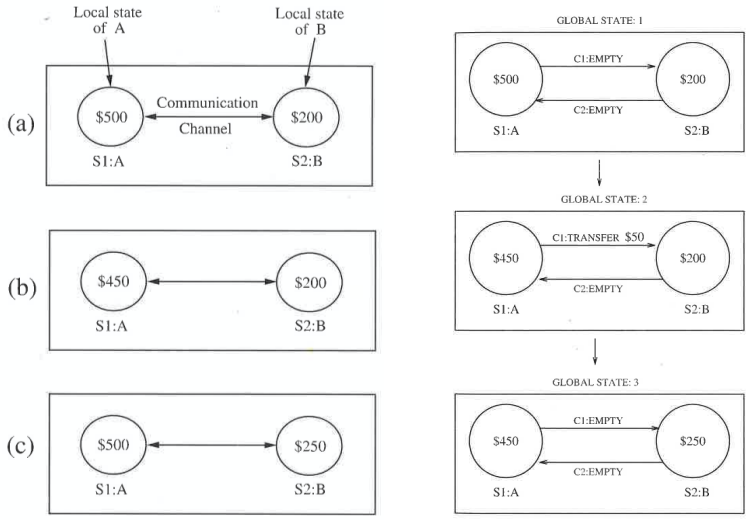
1.3 Global State in a Banking Application

Let S_1 and S_2 be two distinct sites of a distributed system that maintain bank accounts A and B, respectively. Knowledge of the global state of the system may be necessary to compute the net balance of both accounts. Consider an initial local state of the two accounts is \$500 for account A at S_1, and \$200 for account B at S_2; see (a) in Figure 2(a). Now suppose a transfer of \$50 is made by the owner of account A on S_1 to account B. During the collection of a global state, if computer S_1 records the state of A immediately after the debit has occurred, and computer S_2 saves the state of B before the fund transfer message has reached B, then the global state shows \$50 missing; see (b) of Figure 2(a). On the other hand, if A's state is recorded immediately before the transfer and B's state is recorded after account B has been credited \$50, then the global state shows an extra \$50; see (c) of Figure 2(a).

What seems clear from this example is that the state of the communication channels also need to be recorded as part of the global state. Because the channel cannot record its state by itself the sites need to coordinate their state recording activities to record channel state.

Consider again the stages of the \$50 transfer from account A to account B in Figure 2(b). The communication channels C_1 and C_2 are assumed to be FIFO. Suppose A's balance is recorded when the global state is 1 ((a) of Figure 2(b)). Now assume that the global state changes to 2, and the states of the communication channels C_1 and C_2 and of account B are recorded when the global state is 2 ((b) of Figure 2(b)). Then the composite of all the states recorded would show account A's balance as \$500, account B's balance as \$200, and a message in transit to transfer \$50. In other words, an extra \$50 appears in the global state. The reason for this inconsistency is that A's state is recorded before the message is sent and channel C_1's state is recorded after the message is sent. Therefore, a recorded global state may be inconsistent if $n < n'$ where n is the number of messages sent by A along the channel before A's state is recorded and n' is the number of messages sent by A along the channel before the channel's state is recorded.

Suppose channel states are recorded when the global state is 1 and A and B's state are recorded when the global state is 2. Now, the composite state of A, B, and the channel state show a deficit of \$50. This means that the recorded global state may be inconsistent if $n > n'$. Hence a consistent global state requires $n = n'$.



(a) Inconsistent states. (b) Consistent states.

Figure 2: Global states and their transitions in a bank accounts example.

Similarly, we can show that a consistent global state requires $m = m'$ where m' is the number of messages received along the channel before account B's state is recorded and m is the number of messages received along the channel by B before the channel's state is recorded. Since in no system the number of messages sent along the channel can be less than the number of messages received along that channel we have $n' \geq m$. From the two requirements $n = n'$ and $n' \geq m$ we get that a consistent global state must satisfy $n \geq m$. In other words, the state of a communication channel in a consistent global state is the sequence of messages sent along that channel before the sender's state is recorded, excluding the sequence of messages received along that channel before the receiver's state is recorded.

1.4 Checkpointing and Recovery in Concurrent Systems

We now present an algorithm for taking a consistent set of checkpoints (local states), and a rollback recovery algorithm that avoids livelock during recovery from failure [1].

1.4.1 The Checkpoint Algorithm

The checkpoint algorithm takes two kinds of checkpoints on stable storage, permanent and tentative. A *permanent checkpoint* is a local checkpoint at a process and is a part of a consistent global checkpoint. A *tentative checkpoint* is a temporary checkpoint that is made permanent on the successful termination of the checkpoint algorithm. Processes roll back only to their permanent checkpoint.

The checkpoint algorithm assumes that a single process invokes the algorithm. The algorithm has two phases:

Phase 1: An initiating process P_i takes a tentative checkpoint and requests all processes to take tentative checkpoints. Each process informs P_i whether it succeeded in taking a tentative checkpoint. A process says "no" to a request if it fails to take a checkpoint. If P_i learns that all processes have successfully taken a tentative checkpoint, P_i decides that all tentative checkpoints should be made permanent; otherwise, P_i decides that all tentative checkpoints should be discarded.

Phase 2: P_i informs all the processes of the decision it has reached at the end of the first phase. A process, on receiving the message from P_i, will act accordingly. Therefore either all or none of the processes take permanent checkpoints.

The algorithm requires that every process, once it takes a tentative checkpoint not send messages related to the underlying computation until it is informed of P_i's decision.

Messages sent by the checkpointing (or rollback recovery) algorithm are *control messages*. Messages sent as part of the underlying computation are *application messages*. Every outgoing application message m has a field for a label, denoted $m.\ell$. Each process uses monotonically increasing labels in its outgoing application messages. Let $\perp$ denote the smallest label and $\top$ the largest label.

For any two processes X and Y, let m be the last application message that X received from Y after X has taken its last permanent or tentative checkpoint. Then

$$\textit{last\_label\_rcvd}_X[Y] = \begin{cases} m.\ell & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Let m be the first application message X sent to Y after X took its last permanent or tentative checkpoint. Then

$$\textit{first\_label\_sent}_X[Y] = \begin{cases} m.\ell & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Whenever X requests Y to take a tentative checkpoint, X sends $last\_label\_rcvd_X[Y]$ along with its request; Y takes a tentative checkpoint only if

$$last\_label\_rcvd_X[Y] \geq first\_label\_sent_Y[X] > \perp.$$

This condition tells Y that the checkpoint at X has recorded the receipt of one or more application messages sent by Y after Y took its last checkpoint. Therefore Y should take a checkpoint to record those application messages.

Finally, we define $ckpt\_cohort_X = \{Y \mid label\_label\_rcvd_X[Y] > \perp\}$. This set indicates all the processes from which X has received application messages after it has taken its last checkpoint. If X takes a checkpoint, then those processes should also take a checkpoint to record the sending of those application messages. Pseudocode for the checkpointing algorithm is given in Algorithm 1.

1.4.2 The Rollback Recovery Algorithm

The rollback recovery algorithm assumes that a single process invokes the algorithm, and that the checkpoint and the rollback recovery algorithms are not invoked concurrently. The rollback recovery algorithm also has two phases.

Phase 1: An initiating process P_i checks to see if all the processes are willing to restart from their previous checkpoints. A process may reply "no" to a restart request if it is already participating in a checkpointing or a recovering process initiated by some other process. If P_i learns that all the processes are willing to restart from their previous checkpoint, P_i decides that all the processes should restart; otherwise P_i decides that all processes should continue with their normal activities. (P_i may attempt a recovery at a later time.)

Phase 2: P_i propagates its decision to all the processes. On receiving P_i's decision, a process will act accordingly.

The recovery algorithm requires that every process not send messages related to the underlying computation while it is waiting for P_i's decision.

To minimize the number of process rollbacks, the rollback recovery algorithm uses the labelling scheme described in §1.4.1. For any two processes X and Y, let m be the last message that X sent to y before X takes its latest permanent checkpoint. Then

$$\textit{last\_label\_sent}_X[Y] = \begin{cases} m.\ell & \text{if } m \text{ exists} \\ \top & \text{otherwise} \end{cases}$$

When X requests Y to restart from the permanent checkpoint, it sends $last\_label\_sent_X[Y]$ along with its request. Y will restart from its permanent checkpoint only if

$$last\_label\_rcvd_Y[X] > last\_label\_sent_X[Y].$$

When this condition holds, it indicates that X is rolling back to a state where the sending of one or more messages from X to Y is being undone. We also define $roll\_cohort_X = \{Y \mid X \text{can send messages to } Y\}$. Pseudocode for the rollback recovery algorithm is given in Algorithm 2.

Algorithm 1 The Checkpoint Algorithm

Initial state at all processes p:
for all processes q **do**
  $first\_label\_sent_p[q] \leftarrow \perp$;
$OK\_to\_take\_ckpt_p \leftarrow \begin{cases} \text{“}yes\text{”} & \text{if } p \text{ is willing to take a checkpoint} \\ \text{“}no\text{”} & \text{otherwise} \end{cases}$

At initiator process P_i:
for all processes $p \in ckpt\_cohort_{P_i}$ **do**
  send a $Take\_a\_tentative\_ckpt(P_i, last\_label\_rcvd_{P_i}[p])$ message
if all processes replied “yes” **then**
  for all processes $p \in ckpt\_cohort_{P_i}$ **do**
    send $Make\_tentative\_ckpt\_permanent$
else
  for all processes $p \in ckpt\_cohort_{P_i}$ **do**
    send $Undo\_tentative\_ckpt$

At all processes p:
Upon receiving $Take\_a\_tentative\_ckpt(q, last\_label\_rcvd_q[p])$ message from q **do** {
if ($OK\_to\_take\_chpt_p =$ “yes”$)AND(last\_label\_rcvd_q[p] \geq first\_label\_sent_p[q] > \perp$) **then**
  take a tentative checkpoint
  for all processes $r \in ckpt\_cohort_p$ **do**
    send $Take\_a\_tentative\_ckpt(p, last\_label\_rcvd_p[r])$ message
  if all processes $r \in ckpt\_cohort_p$ replied “yes” **then**
    $OK\_to\_take\_ckpt_p \leftarrow$ “yes”
  else
    $OK\_to\_take\_ckpt_p \leftarrow$ “no”
send $(p, OK\_to\_take\_ckpt_p)$ to q
}

Upon receiving $Make\_tentative\_ckpt\_permanent$ message **do** {
Make tentative checkpoint permanent
for all processes $r \in ckpt\_cohort_p$ **do**
  send $Make\_tentative\_ckpt\_permanent$ message
}

Upon receiving $Undo\_tentative\_ckpt$ message **do** {
Undo tentative checkpoint
for all processes $r \in ckpt\_cohort_p$ **do**
  send $Undo\_tentative\_ckpt$ message
}

2 A Distributed Banking Architecture with Checkpointing and Rollbacks

In this project you will implement your own peer-to-peer application program in which processes communicate using sockets to implement CHECKPOINT, a checkpointing and recovery technique for maintaining a consistent global state in a distributed banking application with (simulated) failures.

The architecture of the CHECKPOINT application is illustrated in Figure 3. It shows a bank process for managing customer accounts. Before it does anything else, each customer (peer process) must open an account with the bank.

Customers query the bank to obtain a cohort (subset) C of customers. Each customer $c \in C$ can perform deposits and withdrawals on its own account, and transfers to customers in $C \setminus c$. A customer can also initiate a checkpoint

Algorithm 2 The Rollback Recovery Algorithm

Initial state at all processes p:
$resume\_execution \leftarrow true$
for all processes q **do**
  $last\_label\_rcvd_p[q] \leftarrow \top$;
$willing\_to\_roll_p \leftarrow \begin{cases} \text{“}yes\text{”} & \text{if } p \text{ is willing to roll back} \\ \text{“}no\text{”} & \text{otherwise} \end{cases}$

At initiator process P_i:
for all processes $p \in roll\_cohort_{P_i}$ **do**
  send a $Prepare\_to\_rollback(P_i, last\_label\_sent_{P_i}[p])$ message
if all processes replied “yes” **then**
  for all processes $p \in roll\_cohort_{P_i}$ **do**
    send $Roll\_back$ message
else
  for all processes $p \in roll\_cohort_{P_i}$ **do**
    send $Do\_not\_roll\_back$ message

At all processes p:
Upon receiving $Prepare\_to\_rollback(q, last\_label\_rcvd_{P_i}[p])$ message from q **do** {
if $(willing\_to\_roll_p) AND (last\_label\_rcvd_p[q] > last\_label\_sent_q[p]) AND (resume\_execution_p)$ **then**
  $resume\_execution_p \leftarrow false$
  for all processes $r \in roll\_cohort_p$ **do**
    send $Prepare\_to\_rollback(p, last\_label\_sent_p[r])$ message
  if all processes $r \in roll\_cohort_p$ replied “yes” **then**
    $willing\_to\_roll_p \leftarrow$ “yes”
  else
    $willing\_to\_roll_p \leftarrow$ “no”
  send p $willing\_to\_roll_p$ message to q
}

Upon receiving $Roll\_back$ message AND if $resume\_execution_p = false$ **do** {
restart from p’s permanent checkpoint
for all processes $r \in roll\_cohort_p$ **do**
  send $Roll\_back$ message
}

Upon receiving $Do\_not\_roll\_back$ message **do** {
resume execution
for all processes $r \in roll\_cohort_p$ **do**
  send $Do\_not\_roll\_back$ message
}

operation. To make things interesting, the loss of some transfer messages is simulated. Lost transfers lead to inconsistencies in the amount of money in the bank, hence the system needs to recover by rolling back to a consistent checkpoint of the cohort.

Figure 3 shows a scenario in which six customers have opened accounts with the bank. In this scenario, Brenda, Andy, and Tao are a cohort and Brenda is initiating a checkpoint. Julio and Rajesh are also a cohort and Julio is initiating a rollback to a consistent checkpoint made earlier. For simplicity the intersection of customers in cohorts is empty.

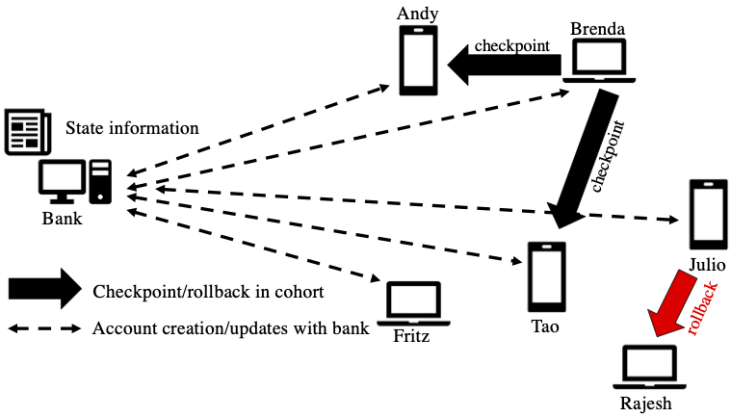


Figure 3: Architecture of the CHECKPOINT application.

3 Requirements of the CHECKPOINT Socket Programming Project

The design and implementation of the CHECKPOINT application involves writing two programs:

1. One program, the `bank`, maintains state information about the customers using CHECKPOINT. The `bank` must be able to process messages issued from a `customer` via a text-based user interface. (No fancy UI is required!) Your `bank` should read one command line parameter specifying the port number (from your range of port numbers) at which it listens for commands. The messages supported by the `bank` are described in §3.1.

2. The second program, the `customer`:

   (a) interacts with the `bank` as a client,
   (b) interacts with itself performing typical banking functions, and
   (c) interacts with other `customer` processes as peers in CHECKPOINT. Your process should read at least two command line parameters, the first is the IPv4 address in dotted decimal notation of the end-host on which the `bank` process the running, and the second is the port number at which the `bank` is listening. The port number should match the port number parameter of the `bank` process. The messages to be supported by a customer interacting with the `bank` as a client are described in §3.1. The messages to be supported by a customer (peer) interacting with itself, and with a cohort as peers are described in §3.2.

Depending on your design decisions (see §4), you may add additional command line parameters to your programs.

3.1 CHECKPOINT: The Bank

Recall that a *protocol* defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event [2].

Every peer-to-peer application requires an always-on process located at a fixed IP address and port to manage the application. In this project, the `bank` is always-on. It maintains a "database" of customers currently running the CHECKPOINT application, and maintains records of the cohorts taking checkpoints, among other information. A single `bank` process must be started before any `customer` processes are run. The `bank` runs in an infinite loop, repeatedly listening for a message on the port bound to the UDP socket sent to it from a `customer`, processes the message, and responds back to the `customer` following the protocol.

To be interesting, at least two `customer` processes must be started. Each one reads commands from `stdin` (again, no fancy UI is expected) until it exits the application. After creating an account with the `bank`, the `customer` may choose to perform typical banking functions. **You must output a well-labelled trace the messages transmitted and received between the `customer` and `bank`, and among `customers` in a cohort, so that the sequence messages sent and received by your CHECKPOINT application, and the events in a checkpoint are clear.**

In the following, angle brackets ⟨⟩ delimit parameters to the command, while the other strings are literals. The `bank` must support messages corresponding to the following commands from a `customer`:

1. `open` ⟨`customer`⟩ ⟨`balance`⟩ ⟨`IPv4-address`⟩ ⟨$\texttt{port}_b$⟩ ⟨$\texttt{port}_p$⟩, to open an account for the named `customer` with the given `balance` with the bank. The `customer` is an alphabetic string of length at most 15 characters giving the customer's name. The `IPv4-address` is the IPv4 address in dotted decimal notation of the end-host running the customer process. This address need not be unique because one or more customer processes may run on the same end-host. $\texttt{port}_b$ is the port number `customer` uses for communication with the bank while $\texttt{port}_p$ is the port number used for communication with other bank customers. The port numbers must come from the set assigned to your group; see §4.3 to find this information.

   Each customer may only have one account. This command returns a return code of `SUCCESS` if the customer is not a duplicate among all those with accounts. In this case, the bank stores a tuple consisting of the customer's name, balance, its IPv4 address and port(s) in a "database." You may implement the "database" however you see fit.

   Otherwise, the bank takes no action and returns a return code of `FAILURE` to the customer indicating failure to open an account due to a duplicate customer name, or any other problem.

2. `new-cohort` ⟨`customer`⟩ ⟨`n`⟩, to obtain a cohort of (integer) $\texttt{n} \geq 2$ customers, including the `customer`. If there are insufficient customers of the bank for a cohort of the given size `n`, or any other reason the cohort cannot be formed, then the bank returns a return code of `FAILURE` to the customer.

   Otherwise, the bank chooses $\texttt{n}-1$ customers at random, which together with the `customer` issuing the request, form a cohort $C, |C| = \texttt{n}$. The bank records the cohort in its database. Cohorts are a subset of bank customers and, for simplicity, have an empty intersection. If you multi-thread your bank process ensure your choice of customers occurs in a critical section.

   The bank forms a returns a list of `n` tuples, with one tuple for each customer $c \in C$, as given in rows 1 through `n` of table below. A tuple for the customer forming the cohort is given in row one of the table. The fields of the tuple are the customer's name, IPv4 address, and port number(s) registered with the customer process.

Customer	Balance	IPv4 Address	Port(s)
$\texttt{customer}_1$	$\texttt{balance}_1$	$\texttt{IP-addr}_1$	$\texttt{port}_{b1}$ $\texttt{port}_{p1}$
$\texttt{customer}_2$	$\texttt{balance}_2$	$\texttt{IP-addr}_2$	$\texttt{port}_{b2}$ $\texttt{port}_{p2}$
⋮	⋮	⋮	⋮
$\texttt{customer}_n$	$\texttt{balance}_n$	$\texttt{IP-addr}_n$	$\texttt{port}_{bn}$ $\texttt{port}_{pn}$

   The list of `n` tuples of cohort information is returned to the `customer` along with a return code to `SUCCESS`.

3. `delete-cohort` ⟨`customer`⟩, to delete the cohort containing the named `customer`. The bank sends a message to each member of the cohort containing the `customer`. This causes each customer in the cohort to delete checkpoints associated with the cohort and to drop any incoming messages from the cohort. Finally, the bank deletes the cohort from its database. This command returns a return code of `SUCCESS` to the initiating the customer if the cohort deletion is successful, and `FAILURE` otherwise. If the deletion of the cohort is successful, the customers are now free to join new cohorts, or exit the system.

4. `exit` ⟨`customer`⟩, to delete the account information associated with the `customer` at the bank. This command returns a return code of `SUCCESS` if the customer account is deleted successfully, and `FAILURE` otherwise; the `customer` process then terminates. The bank deletes the customer's account information.

You may consider providing a mechanism to save the state of the `bank` in a file to facilitate testing and debugging of your application. For example, you may add another command line parameter to your `bank` that reads the state from the named file. This may allow you to bring up your system more quickly.

3.2 CHECKPOINT: The Customers

After creation of a `customer` process in the CHECKPOINT application, it first opens an account at the `bank`; *i.e.*, issue the `open` command at each customer with each customer initializing its state to its `balance`.

Choose one customer process c to issue a `new-cohort` command. Using the list of cohorts C, this process c initializes the cohort. This involves:

1. c sends each member p of the cohort $C \setminus c$ the list of cohort information which p records.
2. Each member $p \in C$ initializes its account balance.
3. Each member $p \in C$ of the cohort runs the initialization of the Checkpoint Algorithm (lines 1–4 of Algorithm 1).
4. Each member $p \in C$ of the cohort runs the initialization of the Rollback Recovery Algorithm (lines 1–5 of Algorithm 2),

3.2.1 Customer Operations

Once the cohort has been initialized, the processes in the cohort may issue any sequence of bank operations (events) until it decides to `exit` the application. These are:

1. `deposit` ⟨`amount`⟩, issued by customer $p \in C$ increments the balance of customer p by `amount`. This `deposit` event is performed locally, *i.e.*, does not involve any message exchange between processes, but is recorded in the state of customer p.

2. `withdrawal` ⟨`amount`⟩, issued by customer $p \in C$ decrements the balance of customer p by `amount`. This `withdrawal` event is performed locally, *i.e.*, does not involve any message exchange between processes, but is recorded in the state of customer p.

3. `transfer` ⟨`amount`⟩ ⟨`q`⟩ ⟨`label`⟩, issued by customer $p \in C$ successfully transfers `amount` from customer p to customer $\mathtt{q} \in C \setminus p$. This operation involves several steps:

   (a) p's balance is decremented by `amount`.
   (b) Increment the label ℓ associated with its outgoing messages to customer `q` in the cohort.
   (c) Send a `transfer` message containing the `amount` being transferred with label ℓ to process `q`.
   (d) Record the transfer operation in the state of p. As described in §1.3, this involves recording the decremented balance at p by `amount` and also the state of the channel from p to `q`, $C_{p\mathtt{q}}$ with the `amount`.
   (e) On receipt of the transfer message by `q` it records its state. This involves recording the incremented balance at `q` by `amount` and also the state of the channel from p to `q`, $C_{p\mathtt{q}}$ as empty.

4. `lost-transfer` ⟨`amount`⟩ ⟨`q`⟩, a transfer from customer $p \in C$ to customer $\mathtt{q} \in C \setminus p$ is simulated as lost. This operation involves several steps:

   (a) p's balance is decremented by `amount`.
   (b) Increment the label ℓ associated with its outgoing messages to customer `q` in the cohort.
   (c) The `transfer` message is only recorded but not sent to `q`. That is, decremented balance at p by `amount` and also the state of the channel from p to `q`, $C_{p\mathtt{q}}$ with the `amount` are both recorded.

   Another transfer from p to `q` will not have the expected label. This is an example of an inconsistency in the state of the banking application. Such a message provides another condition for `q` to rollback.

5. `checkpoint`, issued by customer c runs the Checkpoint Algorithm starting at line 5 of Algorithm 1) with $ckpt\_cohort_c = \{p | label\_label\_rcvd_c[p] > \perp\}$, to take a new permanent checkpoint of the processes in the cohort.

   The checkpointing algorithm should not allow the checkpoint of a `lost-transfer`. It should revert to an earlier checkpoint.

6. `rollback`, issued by customer c runs the Rollback Recovery Algorithm starting at line **??** of Algorithm 2) with $roll\_cohort_c = C \setminus c$, to rollback the execution of the processes in the cohort to a consistent state.

   Note that this operation rolls back to a consistent global state; Any operations that are recorded in the local state are no replayed; they are simply discarded.

Design the protocol to implement the pseudocode of the checkpointing and rollback recovery (Algorithms 1 and 2).

4 Implementation Design Decisions

4.1 Number of Ports to Register and Threading

A customer process communicates with the `bank` as a client, and as a peer with other processes in its cohort in CHECKPOINT. You should set up a separate socket for each such communication.

You may consider using a different thread for handling each socket. Alternatively a single thread may loop, checking each socket one at a time to see if a message has arrived for the process to handle. If you use a single thread, you must be aware that by default the function `recvfrom()` is blocking. This means that when a process issues a `recvfrom()` that cannot be completed immediately (because there is no message to read), the process is put to sleep waiting for a message to arrive at the socket. Therefore, a call to `recvfrom()` will return immediately only if a packet is available on the socket. This may not be the behaviour you want.

You can change `recvfrom()` to be non-blocking, *i.e.*, it will return immediately even if there is no message. This can be done by setting the `flags` argument of `recvfrom()` or by using the function `fcntl()`. See the `man` pages for `recvfrom()` and `fcntl()` in `C/C++` for details; be sure to pay attention to the return codes.

4.2 Defining Message Exchanges and Message Format

As part of this project, you must to define the protocol for CHECKPOINT. This includes defining the format of all messages used between a `bank` and the `customer`, and among peers. This may be achieved by defining a structure with all the fields required by the commands. For example, you could define the command as an integer field and interpret it accordingly. Alternatively, you may prefer to define the command as a string, delimiting the fields using a special character, that you then parse. Indeed, any format is fine so long as you are able to extract the fields from a message and interpret them.

It may also useful to define meaningful return codes to differentiate more specific `SUCCESS` and `FAILURE` states, among other return codes that you may introduce.

4.3 Port Numbers

Both TCP and UDP use 16-bit integer port numbers to differentiate between processes. Both also define a group of well-known ports to identify well-known services. For example, every TCP/IP implementation that supports FTP assigns well-known port of 21 (decimal) to the FTP server.

Clients of these services on the other hand, use ephemeral, or short-lived, ports. These port numbers are normally assigned to the client. Clients normally do not care about the value of the ephemeral port; the client just needs to be certain that the ephemeral port is unique on the client host.

RFC 1700 contains the list of port number assignments from the Internet Assigned Numbers Authority (IANA). The port numbers are divided into three ranges:

- *Well-known ports:* 0 through 1023. These port numbers are controlled and assigned by IANA. When possible, the same port is assigned to a given server for both TCP and UDP. For example, port 80 is assigned for a Web server for both protocols, though all implementations currently use only TCP.
- *Registered ports:* 1024 through 49151. The upper limit of 49151 for these ports is new; RFC 1700 lists the upper range as 65535. These port numbers are not controlled by the IANA. As with well-known ports, the same port is assigned to a given service for both TCP and UDP.

- *Dynamic* or *private ports:* 49152 through 65535. The IANA dictates nothing about these ports. These are the ephemeral ports.

In this project, each group $G \geq 1$ is assigned a set of 500 unique port numbers to use in the following range. If $G \bmod 2 = 0$, *i.e.*, your group number is even, then use the range:

$$\left[\left(\frac{G}{2} \times 1000\right) + 1000, \left(\frac{G}{2} \times 1000\right) + 1499\right]$$

If $G \bmod 2 = 1$, *i.e.*, your group number is odd, then use the range:

$$\left[\left(\left\lceil\frac{G}{2}\right\rceil \times 1000\right) + 500, \left(\left\lceil\frac{G}{2}\right\rceil \times 1000\right) + 999\right]$$

That is, group 1 has range $[1500, 1999]$, group 2 has range $[2000, 2499]$, group 3 has range $[2500, 2999]$, group 4 has range $[3000, 3499]$, and so on.

When you assign ports to processes on the same end-host, they must be unique. If you also have processes running on different end-hosts, you may re-use port numbers because each process is addressed by a pair. It is your responsibility to assign port numbers properly to processes.

Do not use port numbers outside your assigned range, as otherwise you may send messages to another group's bank or customer process by accident causing spurious crashes.

4.4 End-Host Availability

It you are looking for more end-hosts to use in your experimentation, there are several options available:

- Use `generalN.asu.edu` where `N` $= \{3, 4, 5\}$; these have ports numbers > 1024 opened up for us. Use `ifconfig` to determine their IP addresses. These machines use a distributed file system.
- Configure the machines in the racks in BYENG 217 so that they are on the same network; see Lab #1.
- Use CloudLab to configure a virtual network with multiple VMs on one or more sites.

5 Submission Requirements for the Milestone and Full Project Deadlines

All submissions are due before 11:59pm on the deadline date.

1. The milestone is due on Sunday, 02/19/2023. See §5.1 for requirements.
2. The full project is due on Sunday, 03/12/2023. See §5.2 for requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas.

An unlimited number of submissions are allowed. The last submission will be graded.

5.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the `bank`: `open`, `new-cohort` including the initialization of the cohort, `delete-cohort`, and `exit`.

Submit electronically before 11:59pm of Sunday, 02/19/2023 a zip file named `GroupX.zip` where `X` is your group number. **Do not** use any other archiving program except `zip`.

Your `zip` file must contain:

1. **Design document in PDF format (50%).**
   Describe the design of your CHECKPOINT application program *in this order*:

(a) Include a description of your message format for each command implemented for the milestone.
(b) Include a time-space diagram for each command implemented to illustrate the order of messages exchanged between communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event. These diagrams must drawn using a software tool for readability.
(c) Describe your choice of data structures and algorithms used, implementation considerations, and other design decisions.
(d) Include a screen shot showing commits made in your choice of version control system.
(e) Provide a *a link to your video demo* and ensure that the link is accessible to our graders. In addition, provide a list of timestamps in your video at which each step 3(a)-3(g) is demonstrated.

2. **Code and documentation (25%).** Submit well-documented source code implementing the milestone of your CHECKPOINT application.

3. **Video demo (25%).** Upload a video of length at most 5 minutes to YouTube with no splicing or edits *before* the milestone submission deadline. You must provide audio accompaniment to explain your demo.

   The video demo of your CHECKPOINT application for the milestone must include:

   (a) Compile your `bank` and `customer` programs (if applicable).
   (b) Run the freshly compiled programs on at least two (2) distinct end-hosts.
   (c) First, start your `bank` program. Then start three (3) `customer` processes that each `open` an account with the `bank`.
   (d) Have one `customer` issue a `new-cohort` command with $\texttt{n} = 3$.
   (e) Have a different issue the `delete-cohort` command.
   (f) Have each of the customers `exit`.
   (g) Terminate the `bank` process.

   Your video will require at least four (4) windows open: one for the `bank`, and one for each `customer`. Ensure that the font size in each window is large enough to read!

   In addition to your audio accompaniment, you must output a well-labelled trace the messages transmitted and received between `bank` and `customer` processes, as well as other explanatory output, so that the sequence messages sent and received by your CHECKPOINT application is clear.

5.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the `tracker` listed in §3.1. This also involves the design of the protocols between `customer` processes, as described in §3.2.

Submit electronically before 11:59pm of Sunday, 03/12/2023 a zip file named `GroupX.zip` where `X` is your group number. **Do not** use any other archiving program except `zip`.

Your `zip` file must contain:

1. **Design document in PDF format (30%).**
   Extend the design document for the milestone phase of design of your CHECKPOINT application program to include details for the remaining commands implemented for the full project. Provide *in this order*"

   (a) Include a description of your message format for each command designed for `user` process.
   (b) Include a time-space diagram for each command implemented, explaining your design of the checkpointing and rollback recovery algorithms, illustrating the order of messages exchanged between communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.
   (c) Describe your choice of data structures and algorithms used, implementation considerations, and other design decisions.
   (d) Include a screen shot showing commits made in your choice of version control system.
   (e) Provide a *a link to your video demo* and ensure that the link is accessible to graders. In addition, provide a list of timestamps in your video at which each requirement 3(a)-3(h) is satisfied.

2. **Code and documentation (20%).** Submit well-documented source code implementing the milestone of your CHECKPOINT application.

3. **Video demo (50%).** Upload a video of length at most 20 minutes to YouTube with no splicing or edits *before* the full project submission deadline. You must provide audio accompaniment to explain your demo.

   Design an experiment to demonstrate the full functionality of your CHECKPOINT application. It must:

   (a) Compile your `bank` and `customer` programs (if applicable).
   (b) Run the freshly compiled programs on at least four (4) distinct end-hosts.
   (c) Start your `bank` program and then start at least three (3) `customer` processes that each `open` an account with the `bank`.
   (d) Have one `customer` issue a `new-cohort` command with `n` ≥ 3.
   (e) Call the customers c_1, c_2, and c_3. Issue some `deposit` and `withdrawal` operations at some of the customers, followed by successful transfers from c_2 to c_1, and from c_3 to c_1. Then have c_1 issue a checkpoint (these messages should force a checkpoint on all customers).
   (f) Repeat (e) with a different sequence of messages to force another checkpoint on all customers.
   (g) Issue some `deposit` and `withdrawal` operations at customer c_2, followed by a `lost-transfer` to c_1, and then a `transfer` to c_1. Then have c_1 issue a rollback. This should roll back c_2 to its checkpoint before these operations.
   (h) Issue the `delete-cohort` command at one of the customers, then have each of the customers `exit`. Finally, terminate the `bank` process.

   Your video will require at least four (4) windows open: one for the `bank`, and one for each `customer`. Ensure that the font size in each window is large enough to read!

   As before, in addition to your audio accompaniment, you must output a well-labelled trace the messages transmitted and received between `bank` and `customer` processes, and among `customers` processes, including all recording of state information, so that the sequence messages sent and received by your CHECKPOINTCHECKPOINT application is absolutely clear.

References

[1] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 11(1):23–31, January 1987.

[2] James Kurose and Keith Ross. *Computer Networking, A Top-Down Approach*. Pearson, 7th edition, 2017.

[3] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems, Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc., 1994.