Vrije Universiteit Amsterdam

Bachelor Thesis

Implementation of the improved Peterson Kearns rollback algorithm

---

Author: Robbert Alexander Rutte (2655858)

1st supervisor: W.J.Fokkink
2nd reader: supervisor name

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

May 16, 2024

Abstract

In a distributed computer network it is important that the system is resilient and that the data integrity is maintained, furthermore it should be able to handle concurrent failures and have a consistent state when ending execution. One of the protocols that satisfies these criteria is the Peterson-Kearns Rollback Recovery protocol. The algorithm is based on a vector-based clock to be able to establish a causal order of the messages.

In this thesis the improved Peterson-Kearns Rollback Recovery protocol [1] will be implemented and experimented with. It includes a background study, discusses the limitations and challenges for the implementation of the protocol, implements the protocol and measures its overhead and efficiency. The experimentation will include both the memory overhead, communications overhead and the run-time complexity of the implementation.

1 Introduction

In the field of distributed computing, a complex system of processes that participate to accomplish a shared goal, inter-process communication (IPC) is indispensable asset. Yet as crucial as inter-process communication is, it also brings a surplus of complications and complexities that require thorough consideration and planning. From data consistency problems, ensure system resilience to minimizing communications overhead.

The protocol discussed in this thesis aims to tackle the data consistency and system resilience problems. The improved Peterson-Kearns Rollback recovery protocol is able to handle multiple concurrent failures without the program ending in an inconsistent state. In the next section the protocol will be described.

1.1 Improved Peterson Kearns rollback algorithm

The protocol provides data integrity and system resilience through the means of of periodically checkpointing the IPC messages to stable storage. however even when operating in a FIFO channel environment a failure of a process could result in an inconsistent state. For a message's send event could be lost to a failure if it is send causally after the last checkpoint.

In the algorithm optimistic recovery is considered, witch indicates that the checkpointing of the nodes is not synchronized. the rollback procedure handles the uncoordinated nodes reach a consistent state. It will do so by handling the *lost messages*, messages where the receive event was lost in a failure at a crashed process or was rolled back to achieve a consistent state, and *orphaned messages*, messages where the send event was lost in a failure at a crashed node or was rolled back. Once the process initiates the restart or rollback procedure the execution is postponed and the messages that arrived are buffered to be handled once the process has restarted or rolled-back.

The protocol relies on a vector based clock that is used to determine the causal order of the messages. To ensure the consistency of the system the Peterson-Kearns protocol relies on a logical vector clock to calculate witch events are causally after events that are lost in a failure. each of the processes will maintain a clock vector and a failure vector to ensure the the ability to uniquely identify failures. The failure vector ensures a consistent state when processes fail concurrently. An in-depth description of the protocol will be provided in Section II

1.2 Context

Inter-process Communication is a vital part of computer science and facilitates an operating system [2] or a distributed system [3] a way to let different processes communicate and cooperate to accomplish a task. While IPC is inescapable to accommodate cooperation and communication between processes it also numerous problems such as reliability and fault tolerance. A system must be resilient enough to deal with failures to ensure consistency and reliability. The performance overhead must also be optimized since processes that use IPC incur a overhead in computational and memory resources. Furthermore IPC is also inherently more vulnerable to security threats [4], this must be managed to ensure the confidentiality and integrity of the data. A system must also be synchronized to avoid race conditions or a system must have ways to deal with race conditions. If a system uses shared memory across processes this must be well managed to ensure its correctness. IPC is a well established in computer science.

1.3 Problem statement

Distributed systems can always be affected by failures, like processes crashing, software errors or outside factors. All of these will be affecting the system consistency and correctness of the algorithm being executed. To prevent the failures from impacting the system implementing a rollback procedure is crucial. The rollback procedure will ensure that any event that is affected by a failure will be undone and forces the system to resume execution from a previously checkpointed consistent state. This will ensure the correctness and the consistency of the system.

1.4 Objectives

The objective of this thesis is to implement the improved Peterson-Kearns rollback protocol. The protocol should facilitate the system to:

1. Enable its processes to notify others of a failure that occurred.
2. Be able to respond to these failures.
3. Rollback the events affected by the failure.
4. Resume execution from a consistent state.
5. Provide an efficient way to implement the procedure
6. Minimize the communication and performance overhead of normal execution.

1.4.1 Scope

The scope of this thesis includes:

- Implementing the improved Peterson-Kearns protocol.
- Testing the procedure under various scenarios to ensure the correctness of the implementation.
- Assess the performance, memory overhead and the communication overhead.

This Thesis will provide:

- The implementation of the procedure with the source code and documentation.
- The test cases used to evaluate the correctness of the implementation.
- The measurement results and evaluation of the results.
- Indicate areas that could be optimized and enhanced based upon the measurements.

Constraints

- The rollback procedure is to designed to minimize the performance overhead, since this is an optimistic recovery algorithm, under normal operation of the implementation.
- Minimize the memory overhead of the implementation.
- Mimic the *read()* and *write()* c functions to facilitate easy usage of the implementation.
- The rollback procedure should maintain a consistent state.

In section 2 a study of the background of rollback algorithms and related work will be done. Afterwards in section 3 the improved Peterson-Kearns protocol will be explained in detail, the design choices will be explained, the limitations of the implementation will be specified and the challenges of the implementation will be mentioned. In the fourth section the methodology will be described, the research goal and the implementation details will be provided. The results of the experimental findings and analysis of these results will be done in the 5th section as well as an indication where more improvements could be made. The conclusion will be located at the 6th section.

2 Background

Rollback algorithms originate from the database field of computer science, they were first employed to ensure the consistency of a database system and to assure the atomicity and persistence of transactions when failures occur. The types of rollback algorithms that exist are categorized in two segments: *Immediate rollback*, where the algorithm rolls back the affected events immediately after detecting that a failure has occurred and *Deferred rollback*, where the algorithm defers the rollback of affected events and handles them at a later time.

Checkpointing is the periodically saving of events to stable storage. Checkpointing is a central component in every rollback algorithm since when a failure occurs that makes a process crash, it must be reinitialize with part of the memory otherwise to maintain a consistent state the algorithm would need to rollback all events. Therefore to be able to resume from some earlier consistent state checkpoints of the events that occur at all processes need to be saved to stable storage. Checkpoints frequently include the state of the system and previously generated events.

Logging is another critical component of rollback algorithms. A log is kept of the send and receive events that are processed by the process. This log is used when a failure occurs at another process to undo the changes that were made by events that are affected by the failure and restore the state to a consistent state. Logging frequently includes the execution order of the events.

Coordinated checkpoints is where all processes of a system coordinate the checkpointing. In that case the algorithm synchronizes the checkpointing of its processes. They are often used of a way to maintain a consistent global state. This enables the protocol to rollback to a certain checkpoint with all its processes which will achieve a consistent state. No further processing of the checkpointed messages will be necessary since they will not exist.

Since the inception of rollback algorithms they have been applied in many fields other than that of database systems, like in distributed systems and operating systems. Rollback algorithms are vital in databases, operating systems, distributed systems and other fields to handle failures and preserve a consistent state.

2.1 Related Work

in this subsection we will explore existing rollback algorithms and classify them as immediate rollback or deferred rollback.

Immediate rollback

An immediate rollback algorithm is a protocol that will apply changes to the system immediately in addition it will maintain a log file where the applied changes are stored. This often involves committing procedures for every event to ensure the consistency of the system. Once a failure occurs the rollback procedure will undo the changes to the system that have not been committed yet. The major advantage of this method is that the recovery process will not be an expensive procedure, thus this method is usually preferred when failures occur often in a system. Examples of the immediate rollback algorithms are:

Immediate update described in Principles of Transaction Processing [5]. Where an event is either committed when it executes or it is rolled back immediately. These commits are coordinated with other processes. Once a failure occurs the current event is rolled back to ensure a consistent state.

Compensation-based recovery as described in Sagas [6]. In this algorithm when an events affects the state of a system a compensation event is stored in memory. Once a failure occurs the compensation event is executed to revert the state of the system to a consistent state.

Deferred rollback

A deferred rollback algorithm is a protocol that delays the commit procedure of events. This involves the storing of events in a log. Once the commit procedure is complete the changes to the system stored in the log will be applied. Once a failure occurs the rollback procedure will remove affected events in the log and these events will not be affecting the system. The advantages of this method is that the I/O messages will be kept low since multiple events are committed at once, thus this method is usually used if failures will only occur periodically. Examples of deferred rollback algorithms are: *Vector based rollback-recovery* in Rollback based on vector time [7]. This is the original algorithm that the improved Peterson-Kearns recovery-rollback [1] protocol is based upon. In this algorithm causal order of events is provided by a logical vector clock. Once a failure occurs other processes are notified via tokens and the affected events are rolled back to maintain a consistent state.

Distributed transaction processing as described in Checkpointing and Rollback-Recovery for Distributed Systems [8]. In this algorithm a two-phase-commit procedure is used to facilitate the rollback. In the two-phase-commit procedure all processes coordinate the checkpointing to ensure a consistent state.

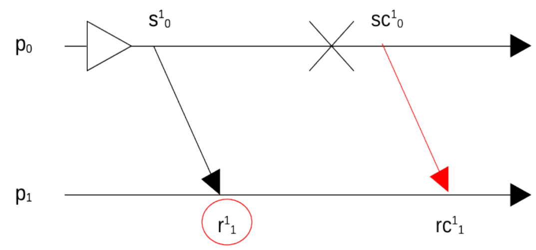
3 Framework

In this section an in-depth overview of the implemented protocol will be given. This will be a summary of the information given in the original Msc thesis of van Eck,C [1].

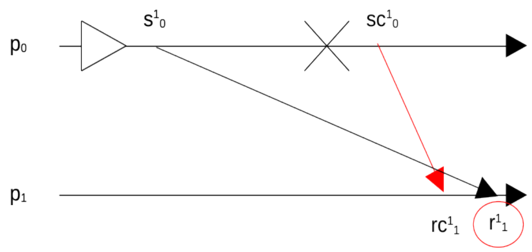
3.1 Classifying events

In this subsection all events will be classified. An event is either a sent or a received message both classified as a send event and a receive event respectively. When these events are affected by a failure they will become either an orphan event or a childless event, these will be further classified into orphan-arrived, orphan-in-transit, childless-lost and childless-orphan events.

3.1.1 Orphan event



(a) Orphan-arrived



(b) Orphan-in-transit

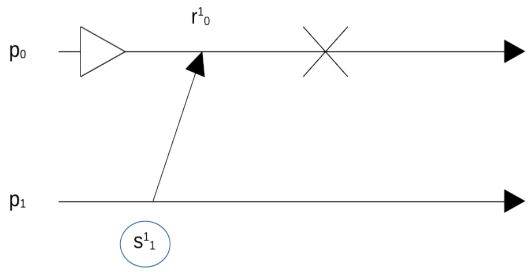
Figure 1: Orphan events

An *orphan event* is created when the send event of a receive event is lost in a failure, this lost send event will be causally after the last checkpoint at the failed process.

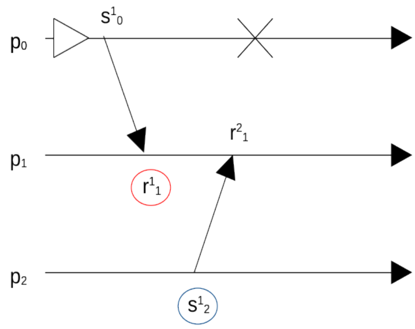
- **Orphan-arrived**: The receive event is causally before the rollback is performed, and is thus before the control message signaling a failure has arrived. Event r^1_1 in figure 1a becomes an orphan-arrived event after the failure at p_0.

- **Orphan-in-transit**: The receive event is causally after the rollback has been performed, and is thus after the control message signaling a failure has arrived. Event r^1_1 in figure 1b is a orphan-in-transit event.

3.1.2 Childless event



(a) Childless-lost



(b) Childless-orphan

Figure 2: Orphan events

A *childless event* is created when the receive event of a send event is lost in a failure, this lost receive event will either be itself lost in the failure or will be lost in the rollback of another process.

- **Childless-lost**: The receive event that is lost is causally after the last checkpoint at the failed process, and is thus itself lost in the failure. Event s^1_1 in figure 2a becomes a childless-lost when the failure at p_0 occurs.

- **Childless-orphan**: The receive event that is lost is causally after an orphaned event at another process, and is thus lost in the rollback procedure at the other process. Event s^1_2 in figure 2b becomes a childless-orphan after the failure at p_0 occurs.

3.2 Algorithm Overview

The improved Peterson-Kearns Rollback recovery protocol [1] has several procedures that are needed for the protocol to function correctly. The protocol employs vector clocks to determine what events are affected by a failure. Each process keeps track of 2 vector clocks, a clock for keeping track of both casual order of the events, the time vector, and the number of times a process has crashed, the failure vector. Both of these clocks will be sent along with a message to inform other processes of the current state of the vector-clock at the sending process. All events will be stored in volatile memory along with its accompanying vector clocks.

3.2.1 Time Vectors

Each process has an associated time vector. The time vector $T = T^0, ..., T^{n-1}$ gets updated whenever an event e_i occurs where n is the number of processes and i is the process id where the event occurred. The T' notation is the time vector before the event e_i occurred.

- When e_i is a send event then the time vector T of a process gets updated to $T^i = T'^i + 1$.
- When e_i is a receive event the time vector T of a process gets updated to $T^j = max(T'^j, e_i.T^j)$ where $j \neq i$ and $T^i = T'^i + 1$

3.2.2 Failure vectors

Each process has an associated failure vector, the failure vector $F = F^0, ..., F^{n-1}$ gets updated when a failure occurs in a process or the process get notified of a failure at another process.

In the case of the failure failure happening at the current process will perform the restart procedure, and therein the failure vector gets updated to $F^i = F'^i + 1$ where i is the id of the current process, afterwards the process sends control messages to all other processes to notify them of its failure. This control message includes:

- *id*: The process ID of the failed process.
- n: The failure number, indicating the amount of failures that occurred in total at the failed process.
- T_i^i: The restart time, indicating the T^i at the moment of restart at the failed process.
- *arrived*: The set of (T^j, F) of the messages received by the failed process that were sent by the process p_j receiving the control message.

In the case that the failure did not occur at the current process, then the current process will be notified of the failure by receiving a control message. When the control message is received the process will perform the rollback procedure, and therein the failure vector will be updated to $F^j = n$ where n is the value received in the control message for the failure number and j is the process ID of the failed process.

3.2.3 Checkpoint procedure

During a checkpoint procedure all information needed to restore the process to a stable state are stored in stable storage. The information to be saved in storage is as follows:

- *state* : the state of a process.
- T: The time vector of the process at the moment the checkpoint occurs.
- *log*: The log where the events that occurred at the process are stored.

During a checkpoint procedure no other event can occur.

3.2.4 Restart procedure

After a failure at a process it will need to restart from the last checkpoint . The operations needed to perform the restart successfully are:

1. The failure vector is updated to $F^i = F'^i + 1$ where i is the ID of the process where the failure occurred.
2. The process will be restored to the latest checkpoint.
3. Send control messages to the other processes.

During a restart procedure no other events can occur.

3.2.5 Rollback Procedure

After a control message is received at a process it will need to perform a rollback to achieve a global consistent state. The operations needed to perform the rollback successfully are:

1. The failure vector is updated to $F^i = F'^i + 1$ where i is the ID of the process where the failure occurred.
2. If some event is orphaned at the process that received the control message, then the state is returned to the last checkpoint that does not include orphaned events. All events after this checkpoint that are not orphaned will be replayed.

3. Any receive events that have been orphaned but have a corresponding send event that was not orphaned will have their receive event replayed.

4. All messages that this process sent to the process where the failure occurred with a non orphaned send event that did not arrive will be re-transmitted.

5. A new checkpoint will be created.

During the rollback procedure no other events can occur.

3.3 Limitations

The limitations of this algorithm are that the maximum number of messages that can be sent by one process will be the upper-limit of the integers used in the time vector, this could be mitigated by allowing the time vector values to roll over back to zero in some way but this would require more logic to accommodate for this.

Another limit of this algorithm would be that when the amount of events stored by a process becomes large and a failure occurs at that process it will need to sent a large set of messages that have arrived at that process. This could be mitigated by occasionally checking what messages have been checkpointed by all processes and then committing these events. After the set of arrived messages will not be needing messages in it that have been committed. This will be explained more in-depth.

If a task needs to be completed that will generate a large amount of event the volatile memory of a process could reach the allowed limit fast. to mitigate this some events could be removed if they have been committed and will not be needed for re-transmission. This would shrink the volatile memory needed for continuous operation. This will be explained more in-depth.

3.3.1 Commit procedure

When a commit procedure is initiated by a process commit messages will be sent between processes. A committed event will be stored in a separate file in stable storage. These messages will be as follows:

1. the process initiating the commit procedure will signal all other processes.

2. When this signal is received at a process it will send the initiating process the T^i of its last *checkpoint* where i is the process ID.

3. After all of the aforementioned messages have been received the initiating process will construct a minimal checkpointed time vector from these values with its T^i of its last *checkpoint*.

4. All events that have a time vector that is $\leq$ the minimal checkpointed time vector can be committed.

5. The initiating process will send this minimal checkpointed time vector to the other processes. Along with the list of committed receive events that have been sent by the recipient of this message, this is needed for the removal of send events.

6. All processes will now be able to commit events, if the process is not the initiator of the commit procedure it will send all other processes a list of of committed receive events that have been sent by the recipient of the message, this is needed for the removal of send events.

in between commit messages other event can occur.

3.3.2 Remove procedure

After all of the other processes have sent the list of committed receive events to a process it can perform the remove procedure. *NextCk(e)* is the checkpoint after the one event e is stored in. the events that can be removed are as follows:

- *Messages stored in volatile storage*: all events e that have *NextCk(e)* $\leq$ the minimal checkpointing time vector can be removed. if e is a sent event its corresponding receive event needs to be contained in the set of committed messages received during the commit procedure.

- *Checkpoints stored in stable storage*: All checkpoints ck_i can be removed if *NextCk(*ck_i $\leq$ the minimal checkpointing time vector.

- *Failures stored in stable storage* : these can only be removed when FIFO channels are used.

3.4 Design Choices

Since one of the requirements of the implementation is that operation without failures occurring should have a small overhead the implementation will be written in C++. In addition to that to ensure fast operation of vector logic the vector package from the C++ standard library was used. To ensure low overhead on the sending and receiving procedures no loops were used in this part of the implementation. The functions to send and receive messages are modeled after the C write and read functions to facilitate simple usage of the implementation.

An implementation will be made for the base protocol without the commit and remove procedures, as well as a version with the commit and remove procedures to test them against each other for a difference in performance.

4 Methodology

4.1 Research Goal

The goal of the experimental setup is that the implementation will be tested on performance. The performance tests that will include the testing performance of both the implementation with and without the commit and remove procedure. For both of these the performance of the sending and receiving of messages will be measured. The performance of the Restart and Rollback procedure will also be measured. The performance of the rollback procedure will also be measured when dealing with concurrent failures from another process. Afterwards the measurements of both of the implementations will be compared.

4.2 Implementation Details

The implementation of the improved Peterson-Kearns rollback recovery protocol is written in C++ version 20 and is provided in appendices A and B.

State class

The State class contains all components needed to maintain a consistent state. the main components of the class are the id of the process, the number of messages in volatile storage, the event log containing the message events, time vector, failure vector, the file descriptors of all other processes(these will be used to resend lost messages), checkpoints and checkpoint time vectors, a set arrived messages and a set of arrived control messages. The version of the class where committing and removing messages is possible the State class also includes a vector to track if the commit or remove procedures can be executed, a set of committed messages, a set of committed receive events form the other processes and a minimal time vector to calculate the messages that can be committed and removed. The most important methods of the State class are listed below.

Public methods:

- **State:** Initializes the class. If the initialization is executed with the restart boolean set to true it will execute the restart operation and sent controll messages to other processes.
- **checkpoint:** Performs a checkpointing procedure .
- **send_msg:** Sends a message and increments the time vector. This will only send messages of type MSG
- **recv_msg:** Receives all message types and handles them accordingly, it will receive normal messages that increments the time vector, it will receive control messages and initiate the rollback procedure, it will receive all commit messages and process them accordingly and it will receive void messages which will not affect the state of the process and will only be passed through to the caller of recv_msg. The method uses c function read to read from its read file descriptor.
- **send_ctrl:** Sends a control message to all other processes to indicate a failure has occurred. The methods uses the c function write to write to the file descriptor of other processes.
- **signal_commit:** Signals to all other processes to start the commit procedure.
- **update_fd:** Updates the file descriptors of a process.

The public methods also include methods to retrieve the stored event log.

In addition to these public methods the class has several private methods that are used by the class to operate. The most important private methods of the State class are listed below:

Private methods:

- **check_duplicate:** Checks if a received message is a duplicate .
- **check_duplicate_ctrl:** Checks if a recieved control message is duplicate.
- **check_duplicate_commit:** Checks if a message has already been committed.
- **check_orphaned:** Checks if a received message is orphaned.
- **rem_log_entries:** Removes log entries from from the event log.
- **rem_checkpoints:** Removes checkpoints.
- **store_msg:** Stores a message in the event log.
- **commit_msgs:** Commits the indicated messages.
- **rollback:** Performs the rollback procedure. This procedure could resend messages that were lost in the failure to the failed process.
- **send_commit:** Sends its own the time vector value to the process that initiated the commit procedure.
- **commit :** Performs the commit procedure and sends the committed receive events to all other processes.
- **remove_data:** Performs the remove procedure.

The private methods also include serialization methods and methods to calculate various pointers and indices. The **send_void** function is included in the namespace Pet_kea. It send a message to a process that does not increment the time vector and is not stored in the event log.

the class is able to send the types of messages listed below:

- **MSG:** This is a message that will be processed by the protocol, it increments the time vector of the process. It includes the time vector and failure vectors at the time of sending, the ID of the sending process and the message size in bytes.
- **CTRL:** This is the message that will be send if a failure has occurred, it will increment the failure vector of the process receiving it. It includes the failure log entry, the ID of the sending process, the amount of messages the sending process has received from the receiving process of the control message and the time vector value and failure vector of these messages. this message can only be sent after a process has restarted.
- **VOID:** This is the message that will not impact either the time vector or the failure vector and will be passed trough to the caller of the **recv_msg** method. And thus not affecting the state of the protocol. This would be used when a message has to be sent between processes that can not alter the state.
- **COMM1:** This is the message that will be sent to initiate the commit procedure
- **COMM2:** This message will contain the its time vector value of its latest checkpoint.
- **COMM3:** This message will contain the combined time vector values of the latest checkpoints of all the processes. These will be used to determine which events can be committed and removed.It also contains the receive events from messages send by the receiver of this COMM message that have been committed by the sender of the COMM message in addition to this the amount of these receive events in the message. This message will be sent by the initiator of the commit procedure only.
- **COMM4:** This message will contain the receive events from messages send by the receiver of this COMM message that have been committed by the sender of the COMM message in addition to this the amount of these receive events in the message. This message will be sent by the non-initiators of the commit procedure only.

The event log consists of an array of structs that contain the send and receive events that are stored, message size, the ID of the message that send the message, the time vector at the moment of sending, the time vector at the moment of receiving and the failure vector at the moment of sending. it also contains the index of the next checkpoint and a boolean indicating if the messages is a send or a receive event.

The checkpointing file in stable storage is organized as follows: at the start of the file an integers will be stored with the amount of total checkpoints in the file. Afterwards the checkpoints will be stored. the header of the checkpoint will consist of two integers and the time vector of the checkpoint. The initial integers will consist of the messages stored in total at the time the checkpoint will be made and the messages stored in total at the checkpoint before this one. The actual checkpoint data will be the serialized msg_log_t.

The committal file will be populated with the serialized msg_log_t of the committed messages.

4.3 Experimental Setup

The experimental setup that was used to measure the overhead cost of the implementation operates as follows:

A parent process creates pipes for its child processes to communicate and then forks them. The child processes will then randomly send each other messages using the State class. The parent process is then able to interrupt one of its children by sending them a SIGINT interrupt. This interrupt is then caught by the child process which will then complete its current select loop and will then exit. After the child has exited the parent will restart the process by forking it again. A process will transmit its new file descriptors over UNIX domain sockets. The child processes will use a select loop to deal with the incoming messages of the other processes.

The experiments will be run on a dell XPS 13 using a intel i7-8565U CPU running at 1.80GHz, the operating system used was Arch-Linux 2024.05.01 release with the included kernel version 6.8.8. Both the implementation of the protocol and the experimental framework is written in C++ version 20.

The measurements are taken using the chrono::high_resolution_clock from time.h. The measurements of the overhead cost for sending and receiving and checkpointing are taken by only measuring how long it took to complete the **send_msg**, **recv_msg** and **checkpoint** methods. For the restart time cost measurements are taken by measuring the time it took for the class to be initialized and transmit all control messages and thus resume normal execution using increasing number of messages stored in stable storage. For the rollback time cost measurements is the time it takes from the moment the process receives the control message until it has handled the rollback procedure and resumed normal operation, each subsequent time using increasing number of messages stored in its event log to determine what effect this has on the rollback procedure. The communication overhead and stable storage memory overhead is calculated using the serialized size of the message and event log. The RAM memory overhead is calculated by the used memory space by the event log struct and the size of the accompanying vectors. The measurements are saved in a CSV file to be able to analyses the results more easily.

The implementation of the experimental setup can be found at GitHub [9].

In addition to the performance test a correctness test will be performed, various scenarios will be checked to indicate the correctness of the implementation.

5 Results and Analysis

5.0.1 Memory overhead

The implementation of the improved Peterson-Kearns protocol both for storing messages in stable storage and in communication utilizes serialized data. The overhead for storing messages in stable storage:

$$3 * sizeof(int) + 3 * sizeof(int) * n \ bytes$$

the communications overhead:

$$3 * sizeof(int) + 2 * sizeof(int) * n \ bytes$$

and the overhead for storing messages in random access memory:

$$96 + 3 * sizeof(int) * n \ bytes$$

where

$$n = number\ of\ processes$$

This shows that when this implementation is used by a large amount of processes that the memory overhead will increase substantially. This overhead could be diminished if an int16_t would be used to store the time and failure vectors, however the downside to this would be that the implementation would be able to handle less messages.

5.1 Experimental Findings

5.1.1 Sending performance

The two figures 3 , 4 show the 200 moving average of time cost of sending a message. The sending cost is tested using different amounts of processes communicating. In the first figure the processes were committing and removing events

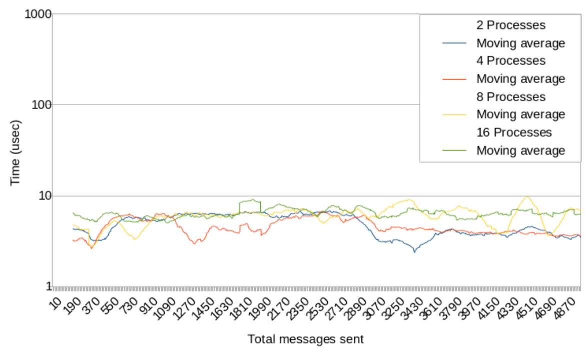


Figure 3: Sending cost for one message : commit procedure (200 MA)

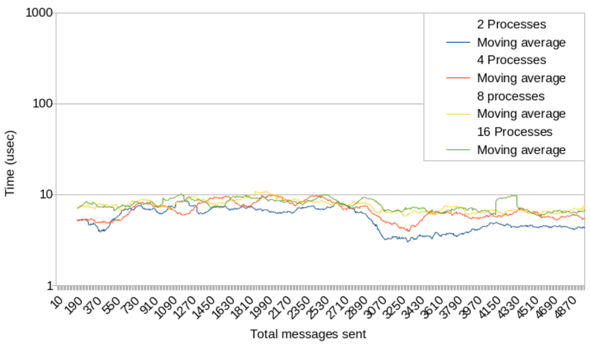


Figure 4: Sending cost for one message (200 MA)

as well as checkpointing the events, in the second figure the processes were only checkpointing the events. This shows that the sending performance is not affected by the increase in processes communicating. It also shows that there is not a big difference in sending cost between weather the messages will be committed or not, however this is to be expected since the operations used for sending a message do not change between the two versions of the protocol.

5.1.2 Receiving performance

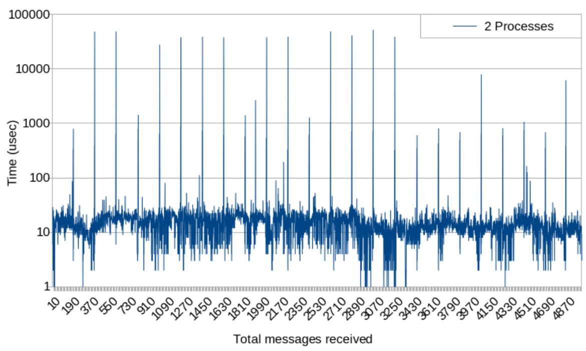


Figure 5: Receiving cost for one message : commit procedure

The two figures 5, 6 above show the receiving cost of receiving a message. The receiving cost is tested using different amounts of processes com-

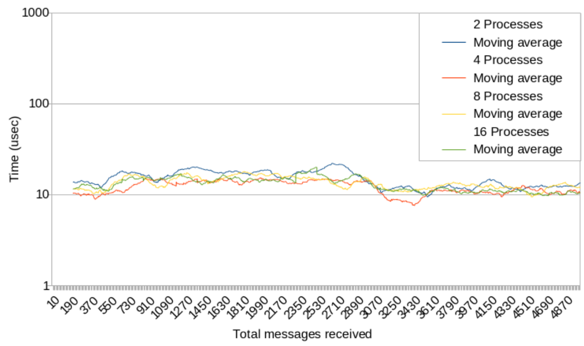


Figure 6: Receiving cost for one message (200MA)

municating. The first figure shows the performance while the processes are committing and removing events. for clarity only the plot for 2 communicating processes is shown. This shows that there are numerous spikes in the time cost if we look at the second figure we can see why this occurs. In the second figure a 200 moving average is used. Here we can see that we do not have such spikes in time cost and can thus conclude that the committing and removing events is a costly procedure. we can also see that the receiving performance is not affected by the increase in processes communicating.

5.1.3 Restart performance

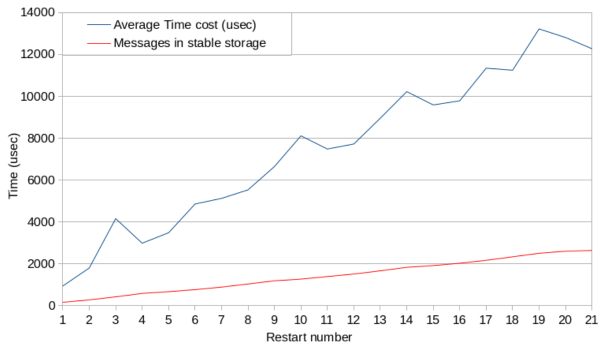


Figure 7: Average restart cost

The figure 7 above shows the time cost of the restart procedure. The blue line shows the time it took from the moment the process called the constructor of the State class until normal operation was resumed, the red line the amount of messages in stable memory that where stored by the process performing the restart. This shows that the time cost of the restart procedure is proportionate to the amount of messages that are stored in stable storage, furthermore it shows that restarting is relatively a cheap procedure.

5.1.4 Rollback performance

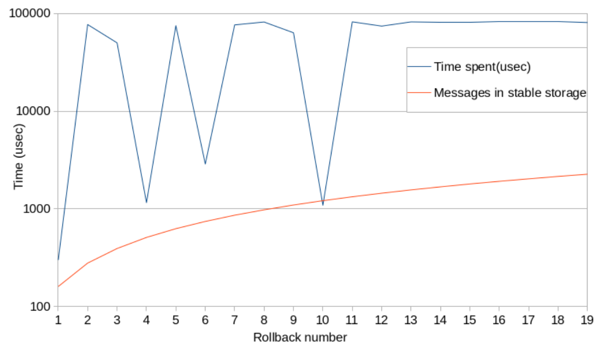


Figure 8: Rollback cost

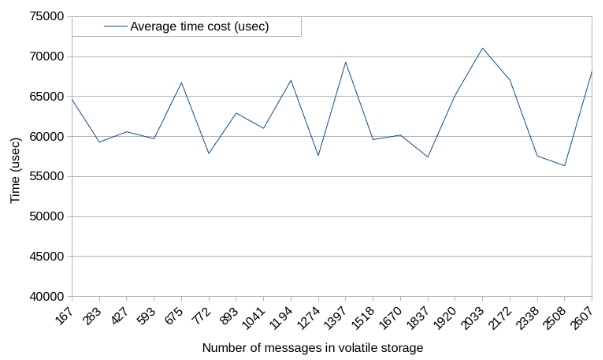


Figure 9: Average rollback cost

The figures 8, 9 above shows the time cost of the rollback procedure. The line in blue shows the time it took from the moment the process received the control message until normal operation was resumed, the red line shows the amount of messages in volatile memory that where stored by the process performing the rollback. This shows that this is a costly procedure since it takes almost 0,1 second most of the time. However at the 1st, 4th, 6th and 10th time the rollback was performed shows a discrepancy, it takes considerably less time to complete the rollback. This is due to that there has not been a checkpoint in between the other process crashing and the process performing the rollback to receive the control message and thus it will not have to remove any checkpoints. The only thing the rollback then does is to replay, resend and remove messages from volatile memory. This indicates that the removal of checkpoints is a costly operation.
In figure 10 the rollback cost when dealing with concurrent failures, all odd number rollbacks are the from the initial failure the even rollback numbers indicate the concurrent failure after the initial one.

This shows that the concurrent failure will cause the rollback to be performed much faster

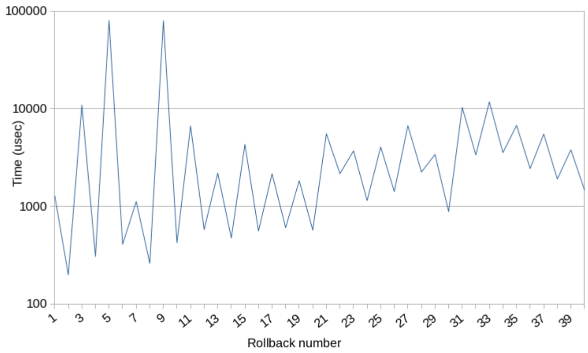


Figure 10: Concurrent rollback cost

than the rollback performed after the initial failure. This is logical since the rollback procedure has already been performed, only message that needed to be resent will again need to be resent, the other parts of the rollback have already been performed at the rollback caused by the initial failure.

5.2 Correctness indication

To provide a Correctness indication a couple of scenarios will be tested:

- *Duplicate*: Duplicate messages will be handled by checking if a set contains the time and failure vectors, afterwards the vectors will be inserted into the set. These will be removed when checkpoints get removed and no more duplicate messages can be sent and is thus not needed any more. The implementation for this is provided in appendix A:93 and A:1372

- *Duplicate-control*: Duplicate control messages will be handled by checking if a set contains its id, failure_number and restart_time, afterwards these values will be added to the set as an entry. The implementation for this is provided in appendix A:106.

- *Orphan-arrived*: Whenever a rollback procedure is called orphaned events will be removed from the message log. An event e_i is orphaned if $e_i.T^i > restart\_time$. if the event e_i is a receive event then $e_i.T^i_{sender} > restart\_time \wedge e_i.F^i_{sender} < failure\_number$. Restart_time and failure_number are the values contained in the control message. The implementation for this is provided in appendix A:880.

- *Orphan-in-transit*: Whenever a message is received the check_orphan method is called.

This method will check if there exists a entry(j, n, T^j) in the failure log that has $msg.F^j < n \wedge msg.T^j > T^j$, if this exists the message is a orphan-in-transit message and will be ignored. the method will be provided in appendix A:129.

- *Childless-lost*: To resolve childless-lost event all send events that are not lost or orphaned but have their receive event lost in the failure will eb re-transmitted. The implementation for this is provided in appendix A:966

- *Childless-Orphaned*: To resolve childless-orphan events all send events that are not lost or orphaned and have their corresponding receive event is orphaned in the rollback procedure will be replayed. The implementation for this is provided in appendix A:931.

This provides a rudimentary correctness indication for the implementation.

5.3 Future Work

For future implementation work the removal of checkpoints could be improved since it now takes a considerable performance penalty to evoke the method. Another point that could be improved is the return value when a process receives a control message and performs the rollback, right now it is just indicated that a rollback has been performed and not what messages have been removed during the rollback. Another improvement that could be made is a more specified error handling with setting the errno correctly. Some changes could also be made to add more constructors to the msg_log_t like the copy and assignment constructors. The serialization of the data to be send and stored could also be handled by some library like Protobuf [10] to facilitate streamlined compatibility with other programming languages. The way committed messages are stored could be altered to a more structured state to provide processes using the committed data an easy understanding of the way the data is stored. Another change that would be welcome is the addition of shadow paging this would increase the fault tolerance of the implementation to ensure data is still valid if a crash occurred in the middle of a write operation.

6 Conclusion

With this we can conclude that the memory overhead does increase proportionate to the number of processes communicating, however we can also conclude that the sending and receiving performance is not greatly affected by the number of processes communicating. The rollback performance as well as the commit receive performance shows that the removing of checkpoints a costly operation. In general is rolling back a costly operation while restarting is a relatively cheap operation.

A pet-kea.cpp

```cpp
#include "pet-kea.hpp"

using namespace std;
typedef unordered_set<vector<int>, vector_hash> uset;

void get_fail_filename(int process_nr, char fail_filename[32])
{
    sprintf(fail_filename, "fail_v_process_%d.dat", process_nr);
}

void get_msg_filename(int process_nr, char msg_filename[32])
{
    sprintf(msg_filename, "msg_process_%d.dat", process_nr);
}

void get_comm_filename(int process_nr, char commit_filename[32])
{
    sprintf(commit_filename, "commit_process%d.dat", process_nr);
}

void get_state_filename(int process_nr, char state_filename[32])
{
    sprintf(state_filename, "state_process%d.dat", process_nr);
}

void Pet_kea::print_msg(struct msg_t *msg)
{
    cout << "MSG  time vector(";
    for (int i = 0; i < (int)msg->time_v.size(); i++)
    {
        cout << msg->time_v[i];
        if (i < (int)msg->time_v.size())
        {
            cout << ", ";
        }
    }
    cout << ") fail vector(";
    for (int i = 0; i < (int)msg->fail_v.size(); i++)
    {
        cout << msg->fail_v[i];
        if (i < (int)msg->fail_v.size())
        {
            cout << ", ";
        }
    }
    cout << ")" << endl;
}

void Pet_kea::print_ctrl_msg(struct ctrl_msg_t *msg)
{
    cout << "CTRL with log(" << msg->log_entry.id << ", " << msg->log_entry.fail_nr << ", ";
    cout << msg->log_entry.res_time << ") messages recieved: " << msg->recieved_cnt << endl;
    for (set<pair<int, vector<int>>>::iterator ptr = msg->recieved_msgs.begin();
         ptr != msg->recieved_msgs.end(); ptr++)
    {
        cout << "       Tj: " << ptr->first << " fail_v: ";
        for (int j = 0; j < (int)ptr->second.size(); j++)
        {
            cout << ptr->second[j] << ":";
        }
        cout << " lenght: " << ptr->second.size() << endl;
    }
}

int *Pet_kea::State::next_checkpoint(int *ptr)
{
    int to_skip = *ptr - *(ptr + 1);
    ptr += 2 + time_v.size();

    for (int i = 0; i < to_skip; i++)
    {
        ptr += ((SER_LOG_SIZE + *ptr) / sizeof(int));
    }
    return ptr;
}

char *Pet_kea::State::get_msg(int i)
{
    return msg_log[i].msg_buf;
}

char **Pet_kea::State::get_msg_log()
{
    char **output_log = (char **)malloc(msg_cnt * sizeof(char *));
    for (int i = 0; i < msg_cnt; i++)
    {
        output_log[i] = msg_log[i].msg_buf;
    }

    return output_log;
}

bool Pet_kea::State::check_duplicate(struct msg_t *msg)
{

    vector<int> merged_time_fail_v;
    merged_time_fail_v = msg->time_v;
    merged_time_fail_v.insert(merged_time_fail_v.end(), msg->fail_v.begin(), msg->fail_v.end());
    if (arrived_msgs.contains(merged_time_fail_v))
        return true;
```

```cpp

    arrived_msgs.insert(merged_time_fail_v);
    return false;
}

bool Pet_kea::State::check_duplicate_ctrl(struct fail_log_t log)
{
    vector<int> fail_log_vector{log.id, log.fail_nr, log.res_time};
    if (arrived_ctrl.contains(fail_log_vector))
        return true;

    arrived_ctrl.insert(fail_log_vector);
    return false;
}

bool Pet_kea::State::check_duplicate_commit(struct msg_log_t *l_msg)
{
    vector<int> merged_time_fail_v;
    merged_time_fail_v = l_msg->time_v_sender;
    merged_time_fail_v.insert(merged_time_fail_v.end(),
                              l_msg->fail_v_sender.begin(), l_msg->fail_v_sender.end());
    if (committed_msg_set.contains(merged_time_fail_v))
        return true;

    committed_msg_set.insert(merged_time_fail_v);
    return false;
}

bool Pet_kea::State::check_orphaned(struct msg_t *msg)
{
    for (int i = 0; i < (int)fail_log.size(); i++)
    {
        if (msg->fail_v[fail_log[i].id] < fail_log[i].fail_nr &&
            msg->time_v[fail_log[i].id] > fail_log[i].res_time)
            return true;
        else
            continue;
    }

    return false;
}

void Pet_kea::State::rem_checkpoints(vector<int> to_remove)
{
    // read file and reconstruct it

    vector<int> reverse_to_remove = to_remove;
    reverse(reverse_to_remove.begin(), reverse_to_remove.end());
    char filename[32];
    get_msg_filename(id, filename);
    msg_out.close();
    ifstream msg_in(filename, ifstream::in | ifstream::binary);
    msg_in.seekg(0, msg_in.end);
    size_t file_size = msg_in.tellg();
    msg_in.seekg(0, ifstream::beg);
    char msg_file[file_size];
    msg_in.read(msg_file, file_size);
    msg_in.close();

    char new_msg_file[file_size];

    int *old_ptr = (int *)msg_file;
    int *new_ptr = (int *)new_msg_file;

    old_ptr++;
    new_ptr++;

    int checkpoint_msg_cnt = 0, checkpoint_last_ckpnt = 0;
    int checkpoint_cnt = (int)checkpoints.size();
    vector<int> new_checkpoints;
    vector<std::vector<int>> new_ck_time_v;
    new_checkpoints.push_back(0);
    new_ck_time_v.push_back(ck_time_v[0]);
    int *temp_ptr;

    for (int i = 1; i < checkpoint_cnt; i++)
    {
        if (!reverse_to_remove.empty() && reverse_to_remove.back() == i)
        {
            reverse_to_remove.pop_back();
            old_ptr = next_checkpoint(old_ptr);
            continue;
        }

        temp_ptr = old_ptr;

        checkpoint_last_ckpnt = checkpoint_msg_cnt;
        checkpoint_msg_cnt += *old_ptr - *(old_ptr + 1);
        *new_ptr = checkpoint_msg_cnt;
        new_ptr++;
        old_ptr++;
        *new_ptr = checkpoint_last_ckpnt;
        new_ptr++;
        old_ptr++;

        temp_ptr = next_checkpoint(temp_ptr);

        memcpy(new_ptr, old_ptr, (temp_ptr - old_ptr) * sizeof(int));

        new_checkpoints.push_back(checkpoint_msg_cnt);
        new_ck_time_v.push_back(ck_time_v[i]);

        new_ptr += (temp_ptr - old_ptr);
```

```
        old_ptr = temp_ptr;
    }

    checkpoints.swap(new_checkpoints);
    ck_time_v.swap(new_ck_time_v);

    size_t test = (size_t)new_ptr;
    size_t test2 = (size_t)(int *)&new_msg_file;

    file_size = new_ptr - (int *)&new_msg_file;
    file_size = test - test2;

    new_ptr = (int *)new_msg_file;

    *new_ptr = checkpoints.size() - 1;

    msg_out.open(filename, ofstream::out | ofstream::binary | ofstream::trunc);
    msg_out.write(new_msg_file, file_size);
    return;
}

int Pet_kea::State::next_checkpoint_after_rem(vector<int> removed_checkpoints, int curr_next_checkpoint)
{
    int result = curr_next_checkpoint;
    for (const int &i : removed_checkpoints)
    {
        if (i < curr_next_checkpoint)
            result--;
        else
            break;
    }
    return result;
}

int Pet_kea::State::rem_log_entries(vector<int> to_remove, int final_index)
{
    cout << id << " old msg cnt:" << final_index << " removing:" << to_remove.size() << endl;
    msg_log_t *new_log = (msg_log_t *)calloc(MAX_LOG, sizeof(msg_log_t));

    vector<int>::iterator curr = to_remove.begin();
    int new_final_index = 0;
    for (int i = 0; i < msg_cnt; i++)
    {
        if (curr != to_remove.end() && i == *curr)
        {
            curr++;
            free(msg_log[i].msg_buf);
            vector<int>().swap(msg_log[i].time_v_reciever);
            vector<int>().swap(msg_log[i].time_v_sender);
            vector<int>().swap(msg_log[i].fail_v_sender);
            continue;
        }
        new_log[new_final_index] = msg_log[i];
        new_final_index++;
        free(msg_log[i].msg_buf);
        vector<int>().swap(msg_log[i].time_v_reciever);
        vector<int>().swap(msg_log[i].time_v_sender);
        vector<int>().swap(msg_log[i].fail_v_sender);
    }
    free(msg_log);
    msg_log = new_log;
    cout << id << " new msg cnt:" << new_final_index << endl;
    return new_final_index;
}

void Pet_kea::State::serialize_commit(struct comm_msg_t *msg, char *data)
{
    int *q = (int *)data;
    *q = (int)msg->msg_type;
    q++;

    *q = msg->sending_process_nr;
    q++;

    switch (msg->msg_type)
    {
    case COMM1:
        *q = 0; // padding
        break;

    case COMM2:
        *q = msg->time_v_j;
        break;

    case COMM3:
        *q = msg->committed_cnt;
        q++;
        for (int i = 0; i < (int)time_v.size(); i++)
        {
            *q = msg->time_v_min[i];
            q++;
        }
        for (uset::iterator ptr = msg->committed_msgs.begin();
             ptr != msg->committed_msgs.end(); ptr++)
        {
            for (int i = 0; i < (int)time_v.size() * 2; i++)
            {
                *q = ptr->at(i);
                q++;
            }
        }
        break;
    case COMM4:
```

```

        *q = msg->committed_cnt;
        q++;
        for (uset::iterator ptr = msg->committed_msgs.begin();
             ptr != msg->committed_msgs.end(); ptr++)
        {
            for (int i = 0; i < (int)time_v.size() * 2; i++)
            {
                *q = ptr->at(i);
                q++;
            }
        }

        break;

    default:
        break;
    }
}

void Pet_kea::State::deserialize_commit(char *data, struct comm_msg_t *msg)
{
    int *q = (int *)data;
    msg->msg_type = (msg_type)*q;
    q++;

    msg->sending_process_nr = *q;
    q++;
    vector<int> temp_vec;
    switch (msg->msg_type)
    {
    case COMM1:
        break;

    case COMM2:
        msg->time_v_j = *q;
        break;

    case COMM3:

        msg->committed_cnt = *q;
        q++;
        for (int i = 0; i < (int)time_v.size(); i++)
        {
            msg->time_v_min.push_back(*q);
            q++;
        }

        for (int i = 0; i < msg->committed_cnt; i++)
        {
            for (int j = 0; j < (int)time_v.size() * 2; j++)
            {
                temp_vec.push_back(*q);
                q++;
            }

            msg->committed_msgs.insert(temp_vec);
            temp_vec.clear();
        }
        break;
    case COMM4:

        msg->committed_cnt = *q;
        q++;
        for (int i = 0; i < msg->committed_cnt; i++)
        {
            for (int j = 0; j < (int)time_v.size() * 2; j++)
            {
                temp_vec.push_back(*q);
                q++;
            }

            msg->committed_msgs.insert(temp_vec);
            temp_vec.clear();
        }

        break;

    default:
        break;
    }
}

void Pet_kea::State::serialize_ctrl(struct ctrl_msg_t *msg, char *data)
{
    int *q = (int *)data;
    *q = (int)msg->msg_type;
    q++;

    *q = msg->recieved_cnt;
    q++;

    *q = msg->sending_process_nr;
    q++;

    *q = msg->log_entry.id;
    q++;
    *q = msg->log_entry.fail_nr;
    q++;
    *q = msg->log_entry.res_time;
    q++;

    for (set<pair<int, vector<int>>>::iterator ptr = msg->recieved_msgs.begin();
```

```
         ptr != msg->recieved_msgs.end(); ptr++)
    {
        *q = ptr->first;
        q++;
        for (int j = 0; j < (int)fail_v.size(); j++)
        {
            *q = ptr->second[j];
            q++;
        }
    }
}

void Pet_kea::State::deserialize_ctrl(char *data, struct ctrl_msg_t *msg)
{
    int *q = (int *)data;
    msg->msg_type = (msg_type)*q;
    q++;

    msg->recieved_cnt = *q;
    q++;

    msg->sending_process_nr = *q;
    q++;

    msg->log_entry.id = *q;
    q++;
    msg->log_entry.fail_nr = *q;
    q++;
    msg->log_entry.res_time = *q;
    q++;

    pair<int, vector<int>> temp_pair;
    for (int i = 0; i < msg->recieved_cnt; i++)
    {

        temp_pair.first = *q;
        q++;
        for (int j = 0; j < (int)fail_v.size(); j++)
        {
            temp_pair.second.push_back(*q);
            q++;
        }

        msg->recieved_msgs.insert(temp_pair);
        temp_pair.second.clear();
    }
}

void Pet_kea::State::serialize(struct msg_t *msg, char *data)
{
    int *q = (int *)data;
    *q = msg->msg_type;
    q++;

    *q = msg->msg_size;
    q++;

    *q = msg->sending_process_nr;
    q++;
    if (msg->msg_type == MSG)
    {
        for (int i = 0; i < (int)time_v.size(); i++)
        {
            *q = msg->time_v[i];
            q++;
        }

        for (int i = 0; i < (int)fail_v.size(); i++)
        {
            *q = msg->fail_v[i];
            q++;
        }
    }

    memcpy(q, msg->msg_buf, msg->msg_size);
}
void Pet_kea::State::deserialize(char *data, struct msg_t *msg)
{
    int *q = (int *)data;
    msg->msg_type = (msg_type)*q;
    q++;

    msg->msg_size = *q;
    q++;

    msg->sending_process_nr = *q;
    q++;
    if (msg->msg_type == MSG)
    {
        for (int i = 0; i < (int)time_v.size(); i++)
        {
            msg->time_v.push_back(*q);
            q++;
        }

        for (int i = 0; i < (int)fail_v.size(); i++)
        {
            msg->fail_v.push_back(*q);
            q++;
        }
    }

    msg->msg_buf = (char *)malloc(msg->msg_size);
```

```
    memcpy(msg->msg_buf, q, msg->msg_size);
}

void Pet_kea::State::serialize_log(struct msg_log_t *log, char *data)
{
    int *q = (int *)data;
    *q = log->msg_size;
    q++;
    *q = log->recipient;
    q++;
    *q = log->process_id;
    q++;
    *q = log->next_checkpoint;
    q++;

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = log->time_v_sender[i];
        q++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = log->time_v_reciever[i];
        q++;
    }

    for (int i = 0; i < (int)fail_v.size(); i++)
    {
        *q = log->fail_v_sender[i];
        q++;
    }

    memcpy(q, log->msg_buf, log->msg_size);
}
int Pet_kea::State::deserialize_log(char *data, struct msg_log_t *log)
{
    int *q = (int *)data;
    log->msg_size = *q;
    q++;
    log->recipient = *q;
    q++;
    log->process_id = *q;
    q++;
    log->next_checkpoint = *q;
    q++;

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        log->time_v_sender.push_back(*q);
        q++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        log->time_v_reciever.push_back(*q);
        q++;
    }

    for (int i = 0; i < (int)fail_v.size(); i++)
    {
        log->fail_v_sender.push_back(*q);
        q++;
    }

    log->msg_buf = (char *)malloc(log->msg_size);
    memcpy(log->msg_buf, q, log->msg_size);

    return (q - (int *)data) * sizeof(int) + log->msg_size;
}

void Pet_kea::State::serialize_state(char *data)
{
    int *q = (int *)data;
    *q = id;
    q++;

    *q = msg_cnt;
    q++;

    *q = arrived_ctrl.size();
    q++;

    *q = committed_msg_set.size();
    q++;

    *q = committed_recieve_events.size();
    q++;

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = time_v[i];
        q++;
    }
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = time_v_min[i];
        q++;
    }
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = commit_v[i];
        q++;
```

```
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        *q = remove_v[i];
        q++;
    }

    for (uset::iterator ptr = arrived_ctrl.begin(); ptr != arrived_ctrl.end(); ptr++)
    {
        *q = ptr->at(0);
        q++;
        *q = ptr->at(1);
        q++;
        *q = ptr->at(2);
        q++;
    }
    for (uset::iterator ptr = committed_msg_set.begin(); ptr != committed_msg_set.end(); ptr++)
    {
        for (int i = 0; i < (int)time_v.size() * 2; i++)
        {
            *q = ptr->at(i);
            q++;
        }
    }
    for (uset::iterator ptr = committed_recieve_events.begin(); ptr != committed_recieve_events.end(); ptr++)
    {
        for (int i = 0; i < (int)time_v.size() * 2; i++)
        {
            *q = ptr->at(i);
            q++;
        }
    }
}

void Pet_kea::State::deserialize_state(char *data)
{
    int *q = (int *)data;
    id = *q;
    q++;

    msg_cnt = *q;
    q++;

    int arrived_ctrl_size = *q;
    q++;

    int committed_msg_set_size = *q;
    q++;

    int committed_recieve_events_size = *q;
    q++;

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        time_v[i] = *q;
        q++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        time_v_min[i] = *q;
        q++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        commit_v[i] = *q;
        q++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        remove_v[i] = *q;
        q++;
    }

    vector<int> temp_vec;
    for (int i = 0; i < arrived_ctrl_size; i++)
    {
        temp_vec.push_back(*q);
        q++;
        temp_vec.push_back(*q);
        q++;
        temp_vec.push_back(*q);
        q++;
        arrived_ctrl.insert(temp_vec);
        temp_vec.clear();
    }

    for (int i = 0; i < committed_msg_set_size; i++)
    {
        for (int j = 0; j < (int)time_v.size() * 2; j++)
        {
            temp_vec.push_back(*q);
            q++;
        }
        committed_msg_set.insert(temp_vec);
        temp_vec.clear();
    }

    for (int i = 0; i < committed_recieve_events_size; i++)
    {
```

```
        for (int j = 0; j < (int)time_v.size() * 2; j++)
        {
            temp_vec.push_back(*q);
            q++;
        }
        committed_recieve_events.insert(temp_vec);
        temp_vec.clear();
    }
}

int Pet_kea::State::send_ctrl()
{
    set<pair<int, std::vector<int>>> recvd_msgs[time_v.size()];

    vector<int> cnt(time_v.size(), 0);
    struct fail_log_t fail_log = {id, fail_v[id], time_v[id]};

    pair<int, vector<int>> temp_pair;
    for (int i = 0; i < msg_cnt; i++)
    {
        if (!msg_log[i].recipient)
        {
            continue;
        }

        temp_pair.first = msg_log[i].time_v_sender[msg_log[i].process_id];
        temp_pair.second = msg_log[i].fail_v_sender;
        recvd_msgs[msg_log[i].process_id].insert(temp_pair);
        cnt[msg_log[i].process_id]++;
    }

    for (int i = 0; i < (int)time_v.size(); i++)
    {
        // prepare and send the ctrl messages
        if (i == id)
            continue;
        struct ctrl_msg_t msg;
        msg.msg_type = CTRL;
        msg.sending_process_nr = id;
        msg.log_entry = fail_log;
        msg.recieved_cnt = cnt[i];
        msg.recieved_msgs = recvd_msgs[i];

        print_ctrl_msg(&msg);

        // send the control message (serialize)
        size_t size = SER_SIZE_CTRL_MSG_T(msg.recieved_cnt, fail_v.size());
        char *data = (char *)malloc(size);
        serialize_ctrl(&msg, data);
        try
        {
            int ret = write(fildes[i][1], data, size);
            if (ret < 0)
            {
                throw runtime_error("failed to write");
            }
        }
        catch (const std::exception &e)
        {
            std::cerr << e.what() << endl;
            perror("write failed");
            return -1;
        }

        free(data);
    }
    return 0;
}

int Pet_kea::State::store_msg(struct msg_t *msg, int recipient)
{
    // store the message
    if (msg_cnt >= MAX_LOG)
    {
        cout << "max log size reached of process " << id << endl;
        return -1;
    }

    msg_log[msg_cnt].msg_size = msg->msg_size;
    msg_log[msg_cnt].next_checkpoint = checkpoints.size();
    msg_log[msg_cnt].msg_buf = (char *)malloc(msg->msg_size);
    memcpy(msg_log[msg_cnt].msg_buf, msg->msg_buf, msg->msg_size);
    msg_log[msg_cnt].time_v_reciever = time_v;
    if (recipient == -1)
    {
        msg_log[msg_cnt].time_v_sender = msg->time_v;
        msg_log[msg_cnt].fail_v_sender = msg->fail_v;
        msg_log[msg_cnt].process_id = msg->sending_process_nr;
        msg_log[msg_cnt].recipient = true;
    }
    else
    {
        msg_log[msg_cnt].time_v_sender = time_v;
        msg_log[msg_cnt].fail_v_sender = fail_v;
        msg_log[msg_cnt].process_id = recipient;
        msg_log[msg_cnt].recipient = false;
    }

    msg_cnt++;
    if (SAVE_CNT != 0 && msg_cnt % SAVE_CNT == 0)
        checkpoint();

    return 0;
```

```
}

int Pet_kea::State::commit_msgs(vector<int> msgs)
{
    char filename[32];
    get_comm_filename(id, filename);

    for (int iterator : msgs)
    {
        char data[msg_log[iterator].msg_size + SER_LOG_SIZE];
        serialize_log(&msg_log[iterator], data);
        commit_out.write(data, msg_log[iterator].msg_size + SER_LOG_SIZE);
    }
    commit_out.flush();

    return 0;
}

int Pet_kea::State::rollback(struct ctrl_msg_t *msg)
{
    cout << "entered the rollback section" << endl;
    print_ctrl_msg(msg);

    // RB.2
    char filename[32];
    get_fail_filename(id, filename);
    ofstream fail_out(filename, ofstream::out | ofstream::binary | ofstream::app);

    struct fail_log_t entry = {msg->sending_process_nr, msg->log_entry.fail_nr, msg->log_entry.res_time};
    fail_out.write((char *)&entry, sizeof(fail_log_t));
    fail_out.close();

    fail_log.push_back(entry);

    //  RB.2.3
    fail_v[msg->sending_process_nr] = msg->log_entry.fail_nr;
    if (time_v[msg->sending_process_nr] > msg->log_entry.res_time)
    {
        //  RB.2.1      remove ckeckpoints T^i > crT^i, set state and T to latest ckeckpoint, replays
        vector<int> checkpoints_to_remove;

        for (int i = 0; i < (int)checkpoints.size(); i++)
        {
            if (ck_time_v[i].at(msg->sending_process_nr) > msg->log_entry.res_time)
                checkpoints_to_remove.push_back(i);
        }

        rem_checkpoints(checkpoints_to_remove);

        // set state and time_v to latestck
        int prev_cnt = msg_cnt;
        msg_cnt = checkpoints.back();
        int temp_msg_cnt = msg_cnt;
        time_v = ck_time_v.back();

        // replay messages
        std::vector<int> indices_to_rem;

        for (int i = msg_cnt; i < prev_cnt; i++)
        {
            if (msg_log[i].time_v_sender[msg->sending_process_nr] <= msg->log_entry.res_time &&
                msg_log[i].time_v_sender[id] >= ck_time_v.back()[id])
            {
                // replay messages only inc time_v and msg_cnt
                cout << "replayed msg" << endl;
                msg_cnt++;
                if (msg_log[i].recipient)
                {
                    time_v[id]++;
                    for (int j = 0; j < (int)time_v.size(); j++)
                    {
                        if (j == id)
                            continue;
                        time_v.at(j) = max(msg_log[i].time_v_sender[j], time_v[j]);
                    }
                }
                else
                {
                    time_v[id]++;
                }
            }
            else if (!msg_log[i].recipient &&
                     msg_log[i].time_v_sender[msg->sending_process_nr] > msg->log_entry.res_time)
            {
                // add to remove vector
                indices_to_rem.push_back(i);
            }
            else if (msg_log[i].recipient &&
                     msg_log[i].time_v_reciever[msg->sending_process_nr] > msg->log_entry.res_time &&
                     msg_log[i].time_v_sender[msg->sending_process_nr] > msg->log_entry.res_time &&
                     msg_log[i].fail_v_sender[msg->sending_process_nr] < msg->log_entry.fail_nr)
            {
                // add to remove vector
                indices_to_rem.push_back(i);
            }
        }
        if (!indices_to_rem.empty())
        {
            msg_cnt = rem_log_entries(indices_to_rem, prev_cnt);
            indices_to_rem.clear();
        }

        //  RB.2.2
```

```
        // RB.3
        prev_cnt = msg_cnt;
        for (int i = temp_msg_cnt; i < prev_cnt; i++)
        {
            // move recv event to the back
            if (msg_log[i].recipient &&
                msg_log[i].time_v_reciever[msg->sending_process_nr] > msg->log_entry.res_time)
            {
                cout << id << " moved RECV event to the back" << endl;
                if (msg_cnt >= MAX_LOG)
                {
                    cout << "max log size reached of process " << id << endl;
                    return -1;
                }

                msg_log[msg_cnt].msg_size = msg_log[i].msg_size;
                msg_log[msg_cnt].recipient = true;
                msg_log[msg_cnt].process_id = msg_log[i].process_id;
                msg_log[msg_cnt].msg_buf = (char *)malloc(msg_log[i].msg_size);
                memcpy(msg_log[msg_cnt].msg_buf, msg_log[i].msg_buf, msg_log[i].msg_size);

                msg_log[msg_cnt].time_v_sender = msg_log[i].time_v_sender;
                msg_log[msg_cnt].fail_v_sender = msg_log[i].fail_v_sender;
                msg_log[msg_cnt].time_v_reciever = msg_log[i].time_v_reciever;

                msg_cnt++;

                time_v[id]++;
                for (int j = 0; j < (int)time_v.size(); j++)
                {
                    if (j == id)
                        continue;
                    time_v.at(j) = max(msg_log[i].time_v_sender[i], time_v[i]);
                }

                // remove the duplicate msg from the log
                indices_to_rem.push_back(i);
            }

            // retransmit send events that have not arrived RB.3.3
            if (!msg_log[i].recipient && msg_log[i].process_id == msg->sending_process_nr &&
                !(msg->recieved_msgs.contains(pair<int, vector<int>>(msg_log[i].time_v_sender[id],
                                                                     msg_log[i].fail_v_sender))))
            {
                cout << id << " retransmitted msg Tj: " << msg_log[i].time_v_sender[id] << "fail_v: ";
                for (int j = 0; j < (int)fail_v.size(); j++)
                {
                    cout << msg_log[i].fail_v_sender[j] << ":";
                }

                cout << " res_time: " << msg_log[i].time_v_sender[msg->sending_process_nr] << endl;
                struct msg_t retransmit_msg;
                retransmit_msg.msg_type = MSG;
                retransmit_msg.sending_process_nr = id;
                retransmit_msg.time_v = msg_log[i].time_v_sender;
                retransmit_msg.fail_v = msg_log[i].fail_v_sender;
                retransmit_msg.msg_size = msg_log[i].msg_size;
                retransmit_msg.msg_buf = (char *)malloc(retransmit_msg.msg_size * sizeof(char));
                memcpy(retransmit_msg.msg_buf, msg_log[i].msg_buf, retransmit_msg.msg_size);

                char data[SER_MSG_SIZE + msg_log[i].msg_size];
                serialize(&retransmit_msg, data);

                // send the message
                try
                {
                    if (write(fildes[msg_log[i].process_id][1], data, SER_MSG_SIZE + msg_log[i].msg_size) < 0)
                        throw runtime_error("failed to write");
                }
                catch (const std::exception &e)
                {
                    std::cerr << e.what() << endl;
                    perror("write failed");
                    return -1;
                }
            }
        }
        if (!indices_to_rem.empty())
        {

            msg_cnt = rem_log_entries(indices_to_rem, msg_cnt);
        }

        checkpoint();
    }
    return 0;
}

Pet_kea::State::State(int process_nr, int process_cnt, int (*fd)[2], bool restart)
    : id(process_nr),
      time_v(process_cnt, 0),
      fail_v(process_cnt, 0),
      msg_cnt(0),
      commit_v(process_cnt, false),
      remove_v(process_cnt, false),
      arrived_msgs()

{
    time_v_min = vector(process_cnt, 0);
    fildes = (int **)malloc(process_cnt * sizeof(int *));
    for (int i = 0; i < process_cnt; i++)
```

```
    {
        fildes[i] = (int *)malloc(2 * sizeof(int));
        fildes[i][0] = fd[i][0];
        fildes[i][1] = fd[i][1];
    }
    if (restart)
    {
        char filename[32];
        get_state_filename(id, filename);
        ifstream state_in(filename, ifstream::in | ifstream::binary);
        state_in.seekg(0, ifstream::end);
        size_t file_size = state_in.tellg();
        state_in.seekg(0, ifstream::beg);
        char state_file[file_size];
        state_in.read(state_file, file_size);
        state_in.close();
        deserialize_state(state_file);

        get_msg_filename(id, filename);
        ifstream msg_in(filename, ifstream::in | ifstream::binary);
        msg_in.seekg(0, msg_in.end);
        file_size = msg_in.tellg();
        msg_in.seekg(0, ifstream::beg);
        char msg_file[file_size];
        msg_in.read(msg_file, file_size);
        msg_in.close();

        int *curr_pos = (int *)msg_file;

        int num_checkpoints = *curr_pos;
        curr_pos++;

        msg_log = (msg_log_t *)calloc(MAX_LOG, sizeof(msg_log_t));

        checkpoints.push_back(0);
        ck_time_v.push_back(vector<int>(process_cnt, 0));

        int read_msg_cnt = 0;
        for (int i = 0; i < num_checkpoints; i++)
        {
            int ck_msg_cnt = *curr_pos;
            checkpoints.push_back(*curr_pos);
            curr_pos++;
            int to_read = ck_msg_cnt - *curr_pos;
            curr_pos++;
            std::vector<int> temp_ck_time_v;

            for (int j = 0; j < (int)time_v.size(); j++)
            {
                temp_ck_time_v.push_back(*curr_pos);
                curr_pos++;
            }
            ck_time_v.push_back(temp_ck_time_v);

            for (int k = 0, ret = 0; k < to_read; k++, read_msg_cnt++)
            {
                ret = deserialize_log((char *)curr_pos, &msg_log[read_msg_cnt]);
                curr_pos += ret / sizeof(int);
            }
        }

        // detect lost messages if crash happened during checkpoint
        if (msg_cnt != read_msg_cnt)
        {
            msg_cnt = read_msg_cnt;
            if (msg_cnt == 0)
            {
                time_v = vector<int>(process_cnt, 0);
            }
            else
            {
                time_v = msg_log[msg_cnt - 1].time_v_sender;
            }
        }

        // insert the arrived messages into arrived_msgs
        vector<int> merged_time_fail_v;

        for (int i = 0; i < msg_cnt; i++)
        {
            if (msg_log->recipient)
            {
                merged_time_fail_v = msg_log[i].time_v_sender;
                merged_time_fail_v.insert(merged_time_fail_v.end(),
                                          msg_log[i].fail_v_sender.begin(), msg_log[i].fail_v_sender.end());
                arrived_msgs.insert(merged_time_fail_v);
            }
        }

        msg_out.open(filename, ofstream::out | ofstream::binary | ofstream::ate | ofstream::in);
        get_comm_filename(id, filename);
        commit_out.open(filename, ofstream::out | ofstream::binary | ofstream::app);
        get_fail_filename(id, filename);

        ifstream fail_in(filename, ifstream::in | ifstream::binary);

        fail_in.seekg(0, fail_in.end);
        file_size = fail_in.tellg();
        fail_in.seekg(0, ifstream::beg);
        char fail_file[file_size];
        fail_in.read(fail_file, file_size);
        fail_in.close();
        curr_pos = (int *)fail_file;
```

```
        fail_log_t temp_fail_log;
        while (curr_pos < (int *)(fail_file + file_size))
        {
            temp_fail_log.id = *curr_pos;
            curr_pos++;
            temp_fail_log.fail_nr = *curr_pos;
            curr_pos++;
            temp_fail_log.res_time = *curr_pos;
            curr_pos++;
            fail_log.push_back(temp_fail_log);
            fail_v[temp_fail_log.id] = max(fail_v[temp_fail_log.id], temp_fail_log.fail_nr);
        }

        ofstream fail_out(filename, ofstream::out | ofstream::binary | ofstream::app);
        fail_out.seekp(0, ofstream::end);
        fail_v[id]++;
        struct fail_log_t entry = {id, fail_v[id], time_v[id]};
        fail_out.write((char *)&entry, sizeof(fail_log_t));
        fail_out.close();

        fail_log.push_back(entry);

        send_ctrl();
    }
    else
    {
        char filename[32];
        get_fail_filename(id, filename);
        ofstream fail_out(filename, ofstream::out | ofstream::binary | ofstream::trunc);
        get_msg_filename(id, filename);
        msg_out.open(filename, ofstream::out | ofstream::binary | ofstream::trunc);
        get_comm_filename(id, filename);
        commit_out.open(filename, ofstream::out | ofstream::binary | ofstream::trunc);

        struct fail_log_t entry = {id, 0, 0};
        fail_out.write((char *)&entry, sizeof(fail_log_t));
        fail_out.close();

        fail_log.push_back(fail_log_t(id, 0, 0));

        msg_log = (msg_log_t *)calloc(MAX_LOG, sizeof(msg_log_t));
        checkpoints.push_back(0);
        ck_time_v.push_back(vector<int>(process_cnt, 0));
    }
    SAVE_CNT = 0;
}

Pet_kea::State::~State()
{
    for (int i = msg_cnt - 1; i >= 0; i--)
    {
        std::vector<int>().swap(msg_log[i].time_v_reciever);
        std::vector<int>().swap(msg_log[i].time_v_sender);
        std::vector<int>().swap(msg_log[i].fail_v_sender);

        free(msg_log[i].msg_buf);
    }

    free(msg_log);
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        free(fildes[i]);
    }

    free(fildes);
    msg_out.close();
    commit_out.close();
}

int Pet_kea::State::checkpoint()
{
    // write state and time vector at the start of the file

    int *update = (int *)malloc(sizeof(int) * 2);
    *update = msg_cnt;
    update++;
    *update = checkpoints.back();
    update--;

    msg_out.seekp(0, ofstream::beg);
    int *num_checkpoints = (int *)malloc(sizeof(int));
    *num_checkpoints = checkpoints.size();
    msg_out.write((char *)num_checkpoints, sizeof(int));
    free(num_checkpoints);

    int *time_v_buffer = (int *)malloc(sizeof(int) * time_v.size());
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        time_v_buffer[i] = time_v[i];
    }

    // append last messages
    msg_out.seekp(0, ofstream::end);

    msg_out.write((char *)update, sizeof(int) * 2);
    free(update);

    msg_out.write((char *)time_v_buffer, sizeof(int) * time_v.size());

    free(time_v_buffer);

    for (int i = checkpoints.back(); i < msg_cnt; i++)
```

```
    {
        char data[msg_log[i].msg_size + SER_LOG_SIZE];
        serialize_log(&msg_log[i], data);
        msg_out.write(data, msg_log[i].msg_size + SER_LOG_SIZE);
    }
    checkpoints.push_back(msg_cnt);
    ck_time_v.push_back(time_v);
    msg_out.flush();

    char filename[32];
    get_state_filename(id, filename);

    ofstream state_out(filename, ofstream::out | ofstream::binary | ofstream::trunc);
    int state_size = SER_STATE_SIZE(arrived_ctrl.size(),
                                    committed_msg_set.size(), committed_recieve_events.size());

    char data[state_size];
    serialize_state(data);

    state_out.write(data, state_size);
    state_out.close();
    return 0;
}

int Pet_kea::send_void(char *input, int fildes[2], int size)
{
    char data[SER_VOID_SIZE + size];

    int *q = (int *)data;
    *q = VOID;
    q++;
    *q = size;
    q++;
    memcpy(q, input, size);

    try
    {
        if (write(fildes[1], data, SER_VOID_SIZE + size) < 0)
            throw runtime_error("failed to write");
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << endl;
        perror("write failed");
        return -1;
    }
    return 0;
}

int Pet_kea::State::signal_commit()
{
    struct comm_msg_t msg;
    msg.msg_type = COMM1;
    msg.sending_process_nr = id;
    char data[SER_COMM1_SIZE];
    serialize_commit(&msg, data);

    // write to all other processes
    try
    {
        for (int i = 0; i < (int)time_v.size(); i++)
        {
            if (i == id)
                continue;
            if (write(fildes[i][1], data, SER_COMM1_SIZE) < 0)
                throw runtime_error("failed to write");
        }
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << endl;
        perror("write failed");
        return -1;
    }
    commit_v[id] = true;
    return 0;
}

int Pet_kea::State::send_commit(int target_id)
{
    struct comm_msg_t msg;
    msg.msg_type = COMM2;
    msg.sending_process_nr = id;
    msg.time_v_j = ck_time_v.back().at(id);

    char data[SER_COMM2_SIZE];
    serialize_commit(&msg, data);
    // write back to sender
    try
    {
        if (write(fildes[target_id][1], data, SER_COMM2_SIZE) < 0)
            throw runtime_error("failed to write");
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << endl;
        perror("write failed");
        return -1;
    }
    return 0;
}

int Pet_kea::State::remove_data()
```

```
{
    vector<int> indices_to_remove, checkpoints_to_remove;
    vector<int> temp_vec;
    unordered_set<std::vector<int>, vector_hash> next_committed_recieve_events;

    // remove checkpoint
    for (int i = 1; i < (int)checkpoints.size() - 1; i++)
    {
        if (ck_time_v[i + 1] <= time_v_min)
        {
            checkpoints_to_remove.push_back(i);
        }
    }

    // remove messages
    for (int i = 0; i < msg_cnt; i++)
    {
        temp_vec.clear();
        temp_vec = msg_log[i].time_v_sender;
        temp_vec.insert(temp_vec.end(), msg_log[i].fail_v_sender.begin(), msg_log[i].fail_v_sender.end());
        if (msg_log[i].next_checkpoint >= (int)checkpoints.size())
        {
            msg_log[i].next_checkpoint = next_checkpoint_after_rem(checkpoints_to_remove,
                                                                   msg_log[i].next_checkpoint);

            if (!msg_log[i].recipient && committed_recieve_events.contains(temp_vec))
            {
                next_committed_recieve_events.insert(temp_vec);
            }
            continue;
        }
        if (msg_log[i].recipient && ck_time_v.at(msg_log[i].next_checkpoint) <= time_v_min)
        {
            // remove form msg_log
            indices_to_remove.push_back(i);
            committed_msg_set.erase(temp_vec);
            if (msg_log[i].fail_v_sender[id] == fail_v[id])
            {
                // remove the entry from the set
                vector<int> merged_time_fail_v;
                merged_time_fail_v = msg_log[i].time_v_sender;
                merged_time_fail_v.insert(merged_time_fail_v.end(),
                                          msg_log[i].fail_v_sender.begin(), msg_log[i].fail_v_sender.end());
                arrived_msgs.erase(merged_time_fail_v);
            }
            continue;
        }

        if (!msg_log[i].recipient && committed_recieve_events.contains(temp_vec))
        {
            if (ck_time_v.at(msg_log[i].next_checkpoint) <= time_v_min)
            {

                // remove from msg_log
                indices_to_remove.push_back(i);
                committed_msg_set.erase(temp_vec);
                continue;
            }
            else
            {
                next_committed_recieve_events.insert(temp_vec);
            }
        }
        msg_log[i].next_checkpoint = next_checkpoint_after_rem(checkpoints_to_remove,
                                                               msg_log[i].next_checkpoint);
    }
    msg_cnt = rem_log_entries(indices_to_remove, msg_cnt);

    rem_checkpoints(checkpoints_to_remove);
    committed_recieve_events.clear();
    committed_recieve_events = next_committed_recieve_events;

    // remove fail_log not possible in asynchronos setting
    return 0;
}

int Pet_kea::State::commit(bool is_instigator)
{
    // commit the messages
    vector<int> committed_msgs;
    for (int i = 0; i < msg_cnt; i++)
    {
        if ((msg_log[i].recipient ? msg_log[i].time_v_reciever : msg_log[i].time_v_sender) <= time_v_min &&
            !check_duplicate_commit(&msg_log[i]))
        {
            committed_msgs.push_back(i);
        }
    }
    commit_msgs(vector<int>(committed_msgs));

    // remove information

    unordered_set<vector<int>, vector_hash> committed_set[time_v.size()];
    struct comm_msg_t msg;
    if (is_instigator)
    {
        msg.msg_type = COMM3;
        msg.time_v_min = time_v_min;
    }
    else
    {
        msg.msg_type = COMM4;
    }
```

```
    msg.sending_process_nr = id;
    vector<int> merged_vector;

    int it;
    while (!committed_msgs.empty())
    {
        it = committed_msgs.back();
        if (msg_log[it].recipient)
        {
            merged_vector = msg_log[it].time_v_sender;
            merged_vector.insert(merged_vector.end(),
                                 msg_log[it].fail_v_sender.begin(), msg_log[it].fail_v_sender.end());
            committed_set[msg_log[it].process_id].insert(merged_vector);
            merged_vector.clear();
        }

        committed_msgs.pop_back();
    }

    remove_v[id] = true;
    int max_cnt = 0;
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        if (max_cnt < (int)committed_set[i].size())
            max_cnt = committed_set[i].size();
    }
    char *data = (char *)malloc(msg.msg_type == COMM3 ? SER_COMM3_SIZE(max_cnt) : SER_COMM4_SIZE(max_cnt));
    // write to all other processes
    for (int i = 0; i < (int)time_v.size(); i++)
    {
        if (i == id)
            continue;

        msg.committed_msgs = committed_set[i];
        msg.committed_cnt = committed_set[i].size();

        if (is_instigator)
        {

            cout << id << " sending comm3 to " << i << "commit cnt: " << msg.committed_cnt << "time_v_min:";
            for (int j = 0; j < (int)time_v.size(); j++)
            {
                cout << msg.time_v_min[j] << ":";
            }
            cout << endl;
        }
        else
        {
            cout << id << " sending comm4 to " << i << "commit cnt: " << msg.committed_cnt << endl;
        }

        serialize_commit(&msg, data);

        if (write(fildes[i][1], data,
                  (msg.msg_type == COMM3 ? SER_COMM3_SIZE(msg.committed_cnt)
                                         : SER_COMM4_SIZE(msg.committed_cnt))) < 0)
            throw runtime_error("failed to write");

        msg.committed_msgs.clear();
    }
    free(data);
    return 0;
}

int Pet_kea::State::send_msg(char *input, int process_id, int size)
{
    // inc T^i
    time_v[id]++;

    struct msg_t msg;

    msg.msg_type = MSG;
    msg.sending_process_nr = id;
    msg.time_v = time_v;
    msg.fail_v = fail_v;
    msg.msg_size = size;
    msg.msg_buf = (char *)malloc(size * sizeof(char));
    memcpy(msg.msg_buf, input, size * sizeof(char));

    char data[SER_MSG_SIZE + size];
    serialize(&msg, data);

    // send the message
    int ret;
    try
    {
        ret = write(fildes[process_id][1], data, SER_MSG_SIZE + size);
        if (ret < 0)
        {
            throw runtime_error("failed to write");
        }
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << endl;
        perror("write failed");
        return -1;
    }

    store_msg(&msg, process_id);

    return 0;
}
```

```
int Pet_kea::State::recv_msg(int fildes[2], char *output, int size)
{
    // read message
    int ret;
    size_t init_read_size = SER_COMM1_SIZE;

    try
    {
        char *extra_data, *data = (char *)malloc(SER_MSG_SIZE + size * sizeof(char));
        extra_data = data;
        ret = read(fildes[0], data, init_read_size);

        if (ret < 0)
            throw runtime_error("failed to read");

        extra_data += init_read_size;

        int *q = (int *)data;

        if (MSG == (msg_type)*q)
        {
            ret = read(fildes[0], extra_data, SER_MSG_SIZE + size - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            struct msg_t msg;
            deserialize(data, &msg);
            free(data);

            if (check_duplicate(&msg))
            {
                return 3;
            }

            if (check_orphaned(&msg))
            {
                return 3;
            }

            time_v[id]++;
            for (int i = 0; i < (int)time_v.size(); i++)
            {
                if (i == id)
                    continue;
                time_v.at(i) = max(msg.time_v[i], time_v[i]);
            }

            // inc T^i and inc T^/j to max(T^j of send event, prev event T^j)

            store_msg(&msg, -1);

            memcpy(output, msg.msg_buf, msg.msg_size);
        }
        else if (CTRL == (msg_type)*q)
        {

            q++;

            char *c_data = (char *)malloc(SER_SIZE_CTRL_MSG_T(*q, fail_v.size()));
            memcpy(c_data, data, init_read_size);

            extra_data = c_data + init_read_size;

            ret = read(fildes[0], extra_data, SER_SIZE_CTRL_MSG_T(*q, fail_v.size()) - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            struct ctrl_msg_t c_msg;
            deserialize_ctrl(c_data, &c_msg);
            free(c_data);
            free(data);
            if (check_duplicate_ctrl(c_msg.log_entry))
            {

                for (int i = 0; i < msg_cnt; i++)
                {
                    if (!msg_log[i].recipient && msg_log[i].process_id == c_msg.sending_process_nr &&
                        !(c_msg.recieved_msgs.contains(pair<int, vector<int>>(msg_log[i].time_v_sender[id],
                                                                                 msg_log[i].fail_v_sender))))
                    {
                        struct msg_t retransmit_msg;
                        retransmit_msg.msg_type = MSG;
                        retransmit_msg.sending_process_nr = id;
                        retransmit_msg.time_v = msg_log[i].time_v_sender;
                        retransmit_msg.fail_v = msg_log[i].fail_v_sender;
                        retransmit_msg.msg_size = msg_log[i].msg_size;
                        retransmit_msg.msg_buf = (char *)malloc(retransmit_msg.msg_size * sizeof(char));
                        memcpy(retransmit_msg.msg_buf, msg_log[i].msg_buf, retransmit_msg.msg_size);

                        char data[SER_MSG_SIZE + msg_log[i].msg_size];
                        serialize(&retransmit_msg, data);

                        // send the message
                        if (write(this->fildes[msg_log[i].process_id][1], data,
                                  SER_MSG_SIZE + msg_log[i].msg_size) < 0)
                            throw runtime_error("failed to write");
                    }
                }

                return 2;
            }

```

```
            rollback(&c_msg);
            return 2;
        }
        else if (VOID == (msg_type)*q)
        {
            q++;

            int v_size = *q;

            char *v_data = (char *)malloc(SER_MSG_SIZE + v_size);
            memcpy(v_data, data, init_read_size);
            free(data);
            extra_data = v_data + init_read_size;

            ret = read(fildes[0], extra_data, SER_VOID_SIZE + v_size - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            // process the void message aka give to method caller
            q = (int *)v_data;
            q++;
            q++;
            memcpy(output, q, v_size);
            free(v_data);
            return 1;
        }
        else if (COMM1 == (msg_type)*q)
        {
            cout << id << " entered COMM1" << endl;
            q++;
            send_commit(*q);
            free(data);
            return 4;
        }
        else if (COMM2 == (msg_type)*q)
        {
            cout << id << " entered COMM2" << endl;

            ret = read(fildes[0], extra_data, SER_COMM2_SIZE - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            struct comm_msg_t comm2_msg;
            deserialize_commit(data, &comm2_msg);
            free(data);
            time_v_min[comm2_msg.sending_process_nr] = comm2_msg.time_v_j;
            commit_v[comm2_msg.sending_process_nr] = true;
            if (commit_v == vector<bool>(time_v.size(), true))
            {
                commit_v.flip();
                time_v_min[id] = ck_time_v.back().at(id);
                commit(true);
            }
            return 4;
        }
        else if (COMM3 == (msg_type)*q)
        {
            cout << id << " entered COMM3" << endl;
            q++;
            q++;
            char *comm3_data = (char *)malloc(SER_COMM3_SIZE(*q));
            memcpy(comm3_data, data, init_read_size);

            extra_data = comm3_data + init_read_size;

            ret = read(fildes[0], extra_data, SER_COMM3_SIZE(*q) - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            struct comm_msg_t comm3_msg;
            deserialize_commit(comm3_data, &comm3_msg);
            free(comm3_data);
            free(data);
            time_v_min = comm3_msg.time_v_min;

            commit(false);
            committed_recieve_events.insert(comm3_msg.committed_msgs.begin(), comm3_msg.committed_msgs.end());
            remove_v[comm3_msg.sending_process_nr] = true;
            return 4;
        }
        else if (COMM4 == (msg_type)*q)
        {
            cout << id << " entered COMM4" << endl;
            q++;
            q++;
            char *comm4_data = (char *)malloc(SER_COMM4_SIZE(*q));
            memcpy(comm4_data, data, init_read_size);

            extra_data = comm4_data + init_read_size;

            ret = read(fildes[0], extra_data, SER_COMM4_SIZE(*q) - init_read_size);
            if (ret < 0)
                throw runtime_error("failed to read");

            struct comm_msg_t comm4_msg;
            deserialize_commit(comm4_data, &comm4_msg);
            free(comm4_data);
            free(data);
            committed_recieve_events.insert(comm4_msg.committed_msgs.begin(), comm4_msg.committed_msgs.end());
            remove_v[comm4_msg.sending_process_nr] = true;
            if (remove_v == vector<bool>(time_v.size(), true))
```

```
            {
                remove_v.flip();
                remove_data();
            }
            return 4;
        }
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << endl;
        perror("failed in recv_msg");
    }

    return 0;
}

int Pet_kea::State::update_fd(int process_id, int fd[2])
{
    if (process_id < 0 || process_id >= (int)time_v.size())
        return -1;

    fildes[process_id][0] = fd[0];
    fildes[process_id][1] = fd[1];
    return 0;
}
```

B pet-kea.hpp

```
/**
 * @file pet-kea.hpp
 * @brief Header file containing the state class and its member functions
 */

#ifndef _PETKEA_HPP_
#define _PETKEA_HPP_

#include <unistd.h>
#include <vector>
#include <set>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstring>
#include <filesystem>
#include <bits/stdc++.h>

const int MAX_LOG = 500;

// Hash function
struct vector_hash
{
    size_t operator()(const std::vector<int>
                          &myVector) const
    {
        std::hash<int> hasher;
        size_t answer = 0;

        for (int i : myVector)
        {
            answer ^= hasher(i) + 0x9e3779b9 +
                      (answer << 6) + (answer >> 2);
        }
        return answer;
    }
};

void get_msg_filename(int process_nr, char msg_filename[32]);

namespace Pet_kea
{
    inline size_t SER_SIZE_CTRL_MSG_T(int recvd_cnt, int v_size)
    {
        return (6 * sizeof(int) + recvd_cnt * (sizeof(int) + v_size * sizeof(int)));
    };
    const int SER_VOID_SIZE = 2 * sizeof(int);

    typedef enum message_type
    {
        MSG,
        CTRL,
        VOID,
        COMM1,
        COMM2,
        COMM3,
        COMM4
    } msg_type;

    struct fail_log_t
    {
        int id;
        int fail_nr;
        int res_time;
        fail_log_t() {}
        fail_log_t(int id, int fail_nr, int res_time) : id(id),
                                                        fail_nr(fail_nr),
                                                        res_time(res_time) {}
    };

    struct comm_msg_t
```

```
    {
        message_type msg_type;
        int sending_process_nr;

        int time_v_j;
        std::vector<int> time_v_min;
        int committed_cnt;
        std::unordered_set<std::vector<int>, vector_hash> committed_msgs;
    };

    struct ctrl_msg_t
    {
        message_type msg_type;
        int sending_process_nr;
        struct fail_log_t log_entry;
        int recieved_cnt;
        std::set<std::pair<int, std::vector<int>>> recieved_msgs;
    };

    struct msg_t
    {
        message_type msg_type;
        int sending_process_nr;
        std::vector<int> time_v;
        std::vector<int> fail_v;
        int msg_size;
        char *msg_buf;
        ~msg_t()
        {
            free(msg_buf);
        }
    };

    struct msg_log_t
    {
        int msg_size;
        bool recipient;
        int process_id;
        char *msg_buf;
        std::vector<int> time_v_sender;
        std::vector<int> time_v_reciever;
        std::vector<int> fail_v_sender;
        int next_checkpoint;
        ~msg_log_t()
        {
            free(msg_buf);
        }
        msg_log_t &operator=(const msg_log_t &other)
        {
            if (this != &other)
            {
                msg_size = other.msg_size;
                recipient = other.recipient;
                process_id = other.process_id;
                time_v_sender = other.time_v_sender;
                time_v_reciever = other.time_v_reciever;
                fail_v_sender = other.fail_v_sender;
                next_checkpoint = other.next_checkpoint;
                char *temp_buf = (char *)malloc(msg_size);
                memcpy(temp_buf, other.msg_buf, msg_size);
                free(msg_buf);
                msg_buf = temp_buf;
            }
            return *this;
        }
    };

    /**
     * @brief Prints the contents of a message.
     * This function prints the contents of a message struct, including message type,
     * sending process number, time vector, failure vector, message size.
     * @param msg Pointer to the message struct to be printed.
     */
    void print_msg(struct msg_t *msg);

    /**
     * @brief Prints the contents of a control message.
     * This function prints the contents of a control message struct, including message type,
     * sending process number, log entry, received count, and received messages.
     * @param msg Pointer to the control message struct to be printed.
     */
    void print_ctrl_msg(struct ctrl_msg_t *msg);

    /**
     * @brief Sends data to another process without it being recorded in the state.
     * @param input Pointer to the data to be sent.
     * @param fildes Array containing file descriptors of the pipe.
     * @param size Size of the data to be sent.
     * @return Number of bytes sent on success, -1 on failure.
     */
    int send_void(char *input, int fildes[2], int size);

    class State
    {
    private:
        int id;
        std::vector<int> time_v;
        std::vector<int> time_v_min;
        std::vector<int> fail_v;
        int **fildes;
        msg_log_t *msg_log;
        int msg_cnt;
        std::vector<bool> commit_v;
```

```cpp
        std::vector<bool> remove_v;
        std::vector<int> checkpoints;
        std::vector<std::vector<int>> ck_time_v;

        std::vector<struct fail_log_t> fail_log;
        std::unordered_set<std::vector<int>, vector_hash> arrived_msgs;
        std::unordered_set<std::vector<int>, vector_hash> arrived_ctrl;
        std::unordered_set<std::vector<int>, vector_hash> committed_msg_set;
        std::unordered_set<std::vector<int>, vector_hash> committed_recieve_events;
        std::ofstream msg_out;
        std::ofstream commit_out;

        /**
         * @brief Checks if a message is a duplicate message.
         * @param msg Pointer to a msg_t structure representing the message to be checked.
         * @return true if if the message is a duplicate, false otherwise.
         */
        bool check_duplicate(struct msg_t *msg);

        /**
         * @brief Checks for duplicate control messages in the fail log.
         * @param log The fail log structure containing control messages to be checked.
         * @return True if duplicate control messages are found, false otherwise.
         */
        bool check_duplicate_ctrl(struct fail_log_t log);

        /**
         * @brief Checks for duplicate commit messages in the message log.
         * @param l_msg Pointer to the message log structure containing commit messages to be checked.
         * @return True if duplicate commit messages are found, false otherwise.
         */
        bool check_duplicate_commit(struct msg_log_t *l_msg);

        /**
         * @brief Checks if a message is orphaned.
         * @param msg Pointer to a msg_t structure representing the message to be checked.
         * @return true if the message is orphaned, false otherwise.
         */
        bool check_orphaned(struct msg_t *msg);

        /**
         * @brief Finds the next_checkpoint variable after removing certain checkpoints.
         * @param removed_checkpoints A vector containing the IDs of checkpoints that have been removed.
         * @param curr_next_checkpoint The ID of the current next_checkpoint variable before removal.
         * @return The ID of the next checkpoint after considering the removed checkpoints.
         */
        int next_checkpoint_after_rem(std::vector<int> removed_checkpoints, int curr_next_checkpoint);

        /**
         * @brief Removes log entries based on provided indices.
         * @param to_remove Vector containing indices of log entries to be removed.
         * @param final_index Index of the final log entry in the log structure.
         * @return New final index
         */
        int rem_log_entries(std::vector<int> to_remove, int final_index);

        /**
         * @brief Removes checkpoints.
         * @param to_remove A vector containing the IDs of checkpoints to be removed.
         */
        void rem_checkpoints(std::vector<int> to_remove);

        /**
         * @brief Retrieves the next checkpoint.
         * @param ptr A pointer to the current checkpoint.
         * @return A pointer to the next checkpoint.
         */
        int *next_checkpoint(int *ptr);

        /**
         * @brief Serializes a commit message.
         * @param msg Pointer to the commit message structure to be serialized.
         * @param data Pointer to the character array where the serialized data will be stored.
         */
        void serialize_commit(struct comm_msg_t *msg, char *data);

        /**
         * @brief Deserializes a commit message.
         * @param data Pointer to the character array containing the serialized commit message.
         * @param msg Pointer to the commit message structure where the deserialized data will be stored.
         */
        void deserialize_commit(char *data, struct comm_msg_t *msg);
        /**
         * @brief Serializes a control message structure into a character array.
         * @param msg Pointer to the control message structure to be serialized.
         * @param data Pointer to the character array where the serialized data will be stored.
         */
        void serialize_ctrl(struct ctrl_msg_t *msg, char *data);

        /**
         * @brief Deserializes a control message from a character array.
         * @param data Pointer to the character array containing serialized data.
         * @param msg Pointer to the control message structure where deserialized data will be stored.
         */
        void deserialize_ctrl(char *data, struct ctrl_msg_t *msg);

        /**
         * @brief Serializes a general message structure into a character array.
         * @param msg Pointer to the message structure to be serialized.
         * @param data Pointer to the character array where the serialized data will be stored.
         */
        void serialize(struct msg_t *msg, char *data);

```

```
        /**
         * @brief Deserializes a general message from a character array.
         * @param data Pointer to the character array containing serialized data.
         * @param msg Pointer to the message structure where deserialized data will be stored.
         */
        void deserialize(char *data, struct msg_t *msg);

        /**
         * @brief Serializes a log message structure into a character array.
         * @param log Pointer to the log message structure to be serialized.
         * @param data Pointer to the character array where the serialized data will be stored.
         */
        void serialize_log(struct msg_log_t *log, char *data);

        /**
         * @brief Deserializes a log message from a character array.
         * @param data Pointer to the character array containing serialized data.
         * @param log Pointer to the log message structure where deserialized data will be stored.
         * @return returns the size of the deserialized log message
         */
        int deserialize_log(char *data, struct msg_log_t *log);

        /**
         * @brief Serializes the state data.
         * @param data A pointer to the character array where the serialized state data will be stored.
         */
        void serialize_state(char *data);

        /**
         * @brief Deserializes the state data.
         * @param data A pointer to the character array containing the serialized state data.
         */
        void deserialize_state(char *data);

        /**
         * @brief Stores a general message.
         * @param msg Pointer to the message structure to be stored.
         * @param recipient     -1 if the stored msg is a recieve event,
         *                      any positive interger for the recipient process_id.
         * @return 0 on success, -1 on failure.
         */
        int store_msg(struct msg_t *msg, int recipient);

        /**
         * @brief Commits a list of messages.
         * @param committed_msgs A vector containing the IDs of messages to be committed.
         * @return Returns 0 if the messages are successfully committed.
         *         Returns a non-zero value if an error occurs.
         */
        int commit_msgs(std::vector<int> committed_msgs);

        /**
         * @brief performs a rollback
         * @param msg Pointer to the  control message structure that activated the rollback
         * @return 0 on success, -1 on failure.
         */
        int rollback(struct ctrl_msg_t *msg);

        /**
         * @brief Sends a COMM2 message to the target with the specified ID.
         * @param target_id The ID of the target to which the commit message will be sent.
         * @return  Returns 0 if the commit message is successfully sent.
         *          Returns a non-zero value if an error occurs.
         */
        int send_commit(int target_id);

        /**
         * @brief Commits all messages that can be safely committed
         * @param is_instigator A boolean value indicating whether the calling function
         *                      is the initiator of the commit procedure (true) or not (false).
         * @return Returns 0 if the action is successfully committed.
         *         Returns a non-zero value if an error occurs.
         */
        int commit(bool is_instigator);

        /**
         * @brief Removes all messages and checkpoints that can safely be removed
         * @return Returns 0 if the data is successfully removed.
         *         Returns a non-zero value if an error occurs.
         */
        int remove_data();

    public:
        const int SER_LOG_SIZE = 4 * sizeof(int) + time_v.size() * 3 * sizeof(int);
        const int SER_MSG_SIZE = 3 * sizeof(int) + time_v.size() * 2 * sizeof(int);
        const int SER_COMM1_SIZE = 3 * sizeof(int);
        const int SER_COMM2_SIZE = 3 * sizeof(int);
        inline int SER_COMM3_SIZE(int committed_cnt)
        {
            return (committed_cnt * time_v.size() * 2) * sizeof(int) + (3 + time_v.size()) * sizeof(int);
        };
        inline int SER_COMM4_SIZE(int committed_cnt)
        {
            return (committed_cnt * time_v.size() * 2) * sizeof(int) + 3 * sizeof(int);
        };
        inline int SER_STATE_SIZE(int arr_ctrl_size, int comm_msg_size, int comm_recv_events_size)
        {

            return (5 + (time_v.size() * 4) + ((comm_msg_size + comm_recv_events_size) * time_v.size() * 2) + (3 * arr_ctrl_size)) * sizeof(int);
        };

        /**
```

```cpp
     * @brief   Initalizes to 0, when 0 no automatic checkpointing will be executed,
     *          otherwise if (NUM_stored_events % SAVE_CNT == 0) is true a checkpoint is made
     */
    int SAVE_CNT;

    // Copy assignment operator
    State &operator=(const State &other)
    {
        if (this != &other)
        {
            id = other.id;
            time_v = other.time_v;
            fail_v = other.fail_v;

            if (fildes)
            {
                for (int i = 0; i < (int)time_v.size(); i++)
                {
                    free(fildes[i]);
                }
                free(fildes);
            }
            fildes = (int **)malloc((int)time_v.size() * sizeof(int *));
            for (int i = 0; i < (int)time_v.size(); i++)
            {
                fildes[i] = (int *)malloc(2 * sizeof(int));
                fildes[i][0] = other.fildes[i][0];
                fildes[i][1] = other.fildes[i][1];
            }

            msg_cnt = other.msg_cnt;
            msg_log_t *temp_log = (msg_log_t *)calloc(MAX_LOG, sizeof(msg_log_t));

            for (int i = 0; i < other.msg_cnt; i++)
            {
                temp_log[i] = other.msg_log[i];
            }
            for (int i = other.msg_cnt - 1; i >= 0; i--)
            {
                std::vector<int>().swap(msg_log[i].time_v_reciever);
                std::vector<int>().swap(msg_log[i].time_v_sender);
                std::vector<int>().swap(msg_log[i].fail_v_sender);

                free(msg_log[i].msg_buf);
            }
            free(msg_log);
            msg_log = temp_log;

            checkpoints = other.checkpoints;

            ck_time_v = other.ck_time_v;

            fail_log = other.fail_log;

            arrived_msgs = other.arrived_msgs;

            arrived_ctrl = other.arrived_ctrl;

            msg_out.close();
            char filename[32];
            get_msg_filename(id, filename);
            msg_out.open(filename,
                         std::ofstream::out | std::ofstream::binary | std::ofstream::ate | std::ofstream::in);
        }
        return *this;
    }

    /**
     * @brief Constructor for State class.
     * @param process_nr The process number.
     * @param process_cnt The total number of processes.
     * @param fd The file descriptors from all processes.
     * @param restart Flag indicating whether the process is being restarted.
     */
    State(int process_nr, int process_cnt, int (*fd)[2], bool restart);

    /**
     * @brief Destructor for State class.
     */
    ~State();

    /**
     * @brief Retrieves a message buffer.
     * This function retrieves the message buffer at the specified index `i`.
     * @param i Index of the message buffer to retrieve.
     * @return Pointer to the message buffer.
     */
    char *get_msg(int i);

    /**
     * @brief Retrieves the message log.
     * This function retrieves the message log, which is an array of message buffers.
     * @return Pointer to an array of message buffers.
     */
    char **get_msg_log();

    /**
     * @brief Creates a checkpoint of the process state and the recieved messages.
     * @return 0 on success, -1 on failure.
     */
    int checkpoint();

```

```
        /**
         * @brief Signals a commit procedure to take place. the process that calls this function is the
             instigator
         *        of the commit procedure.
         * @return Returns 0 upon successful signaling of the commit.
         *         Returns a non-zero value if an error occurs during signaling.
         */
        int signal_commit();

        /**
         * @brief Sends a message to another process that will be recorded in the state.
         *
         * @param input Pointer to the message to be sent.
         * @param fildes Array containing file descriptors of the pipe.
         * @param size Size of the message to be sent.
         * @return Number of bytes sent on success, -1 on failure.
         */
        int send_msg(char *input, int process_id, int size);

        /**
         * @brief Receives a message from another process
         * @param fildes Array containing file descriptors of the pipe.
         * @param output Pointer to the buffer where the received message will be stored.
         * @param size Size of the buffer.
         * @return 0 on success, 1 after recieving a VOID msg, 2 after recieving a CTRL msg,
         *         3 after recieving a duplicate or orphaned message that was discarded, -1 on failure.
         */
        int recv_msg(int fildes[2], char *output, int size);

        /**
         * @brief Sends control message to other processes to indicate that a failure occured.
         * @param fildes Array of arrays containing file descriptors of the pipes.
         */
        int send_ctrl();

        /**
         * @brief updates file descriptors
         * @param process_id The process id of the file descriptors to be changed.
         * @param fd The new file descriptors.
         * @return 0 on success, -1 on failure.
         */
        int update_fd(int process_id, int fd[2]);
    };
}

#endif
```

References

[1] C. van Ek, "Optimistic recovery protocol for concurrent failures," Ph.D. dissertation, Universiteit van Amsterdam, 2023.

[2] W. R. Stevens, *UNIX Network Programming, Volume 2: Interprocess Communications*. Prentice Hall, 1999.

[3] Anonymous, "Interprocess communication in distributed systems," *GeeksforGeeks*, 2019, accessed: 2024-05-16. [Online]. Available: https://www.geeksforgeeks.org/interprocess-communication-in-distributed-systems/

[4] ——, "Inter-process communication, technique t1559," *MITRE ATT&CK*, 2024, accessed: 2024-05-16. [Online]. Available: https://attack.mitre.org/techniques/T1559/

[5] P. A. Bernstein and E. Newcomer, *Principles of transaction processing*. Morgan Kaufmann, 2009.

[6] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[7] S. L. Peterson and P. Kearns, "Rollback based on vector time," in *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*. IEEE, 1993, pp. 68–77.

[8] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[9] R. A. Rutte, "Implementation of the peterson-kearns rollback recovery protocol," 2024. [Online]. Available: https://github.com/robybert/Peterson_kearns_rollback_algorithm

[10] G. LLC, "Protobuf," 2024. [Online]. Available: https://protobuf.dev/