# 15-440 Spring 2025
# Project 4: Two-phase Commit for Group Photo Collage

**Important Dates:**

| | |
|---|---|
| Project Handout: | Tuesday April 8, 2025 |
| Checkpoint 1 due: | Tuesday April 15, 2025, 11:59 PM EST |
| Final Due: | Tuesday April 22, 2025, 11:59 PM EST |
| Submission Limits: | **10 Autolab submissions** per checkpoint without penalty (5 additional with increasing penalty) |

## You will learn to:

1. Implement a distributed transaction using two-phase commit
2. Deal with lost and delayed messages
3. Handle and recover from node crashes and network failures
4. Utilize logging to persistent storage for failure recovery

## Introduction

At the dawn of photography, in the late 19th and early 20th century, group photos were a way of capturing the spirit of an important occasion in the lives of people. The technology was primitive by today's standards: cameras were large and bulky, it took many seconds or tens of seconds to capture a picture, the chemical processes were slow and messy, and so on. At the same time, you could be sure that a group photo accurately captured reality since it was very difficult to forge the content of photographs. A group photograph was thus a true record of reality. A group photo of the Solvay conference in 1927 (Figure 1), shows Albert Einstein, Marie Curie, Paul Dirac, Werner Heisenberg, and many other familiar names from your physics textbooks.

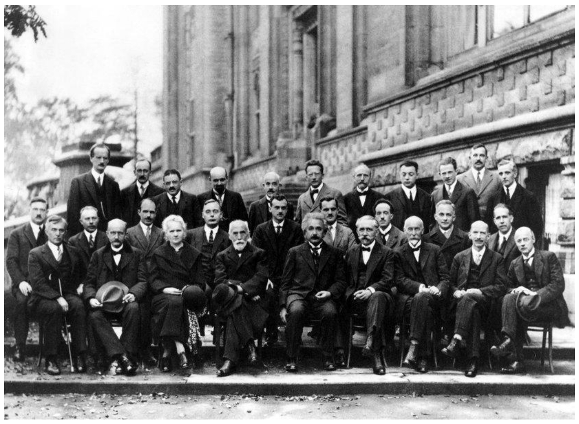
Figure 1. Solvay conference

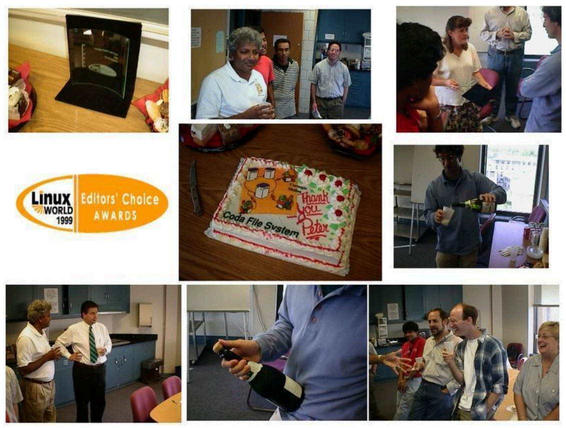

Figure 2. LinuxWorld award

Fast forward to 2025. Capturing images is no longer limited to a few professional photographers. Your smartphone can take excellent color pictures and videos, and you can see and share them right away without film or processing. Thousands of images may be captured and shared at large gatherings by hundreds of participants. On the minus side, you can use GIMP or Photoshop to alter images easily and virtually undetectably. You can post the forged images to social media and embarrass people or ruin their careers and lives. Furthermore, copyright and licensing issues become complex when multiple images are combined.

While mobile computing and imaging technology has changed beyond recognition, people have not changed much. They still love to record important and fun occasions in their lives, spent in the company of their friends. A group photo in the 21st century looks very different (Figure 2), but is no less important in the lives of people.

In this project, you will build a part of a system that generates and publishes group collages assembled from multiple images contributed by multiple individuals. The system consists of several UserNodes, each of which represents a single person's smartphone or laptop, and a single Server, which coordinates the publication of collages. The process for publishing a collage follows these steps:

1. Someone constructs a candidate collage made from images shared by the UserNodes, and posts it to the Server.
2. The Server initiates a two-phase commit procedure, letting the users that contributed images to the collage examine it to see if they are happy with the result.
3. UserNodes either approve or disapprove of it.
4. Due to licensing terms on the published collages, a UserNode needs to ensure that any of its source images appear in no more than one committed / published collage.
5. Only if all UserNodes that contribute an image to the collage approve (i.e., the two-phase commit succeeds) the collage is published (written to the Server's working directory).
6. A successful commit must produce particular side effects on the UserNodes as well. A UserNode is required to stop sharing (i.e., remove from its working directory) any of its images that appear in a published collage.

This system needs to be robust to lost messages and to node failures / reboots. Furthermore, your system needs to be able to process multiple collage commits concurrently.

## Requirements/Deliverables

### We will provide:

- We provide several components for this project. These require that the project be implemented in Java, in a Unix / Linux environment, e.g., Andrew Unix servers.
- We will provide a Java class (Project4) that sets up the working environment and launches the system components (Server and UserNodes) as separate processes.
- We will provide you a Java class (ProjectLib) that provides a simple UDP-like datagram messaging service, a mechanism to get the next candidate collage (at the Server), and a method to ask a user if they approve the image (at a UserNode).

## You will create:

- You will create a class called “Server” that implements a main() method. The Server will coordinate and perform two-phase commits on candidate collage images.
- You will create a class called “UserNode” that implements a main() method. The UserNodes will participate in the two-phase commit started by the Server.

## Your code should do the following:

- Your Server class and UserNode class will implement main() methods. A single instance of Server and multiple instances of UserNode will be run by the Project4 class as separate processes at the beginning of a test.
- Your Server will be provided a single command line argument: <port>. This specifies the port used by the Project4 class, and needs to be provided to the constructor of ProjectLib. Your Server will be assigned “Server” as its “address” for messaging.
- Your UserNode will be provided two command-line arguments: <port> and <id>. <port> specifies the port used by the Project4 class. <id> is the “address” of this UserNode instance. Both of these are passed to the constructor of ProjectLib.
- Your Server class, or another class it uses, needs to implement the CommitServing interface. This interface defines a single method (startCommit) that will be called to inform your Server that a new candidate collage has been posted, and to indicate which source images contributed to it. This should cause a new two-phase commit operation to begin.
- Both the Server and UserNode should instantiate ProjectLib. The Server should call the ProjectLib constructor with <port> (provided on the command line) and the CommitServing object reference as parameters. The Server will automatically be assigned an address of “Server”. The UserNodes should call the ProjectLib constructor with <port> and <id> (both from the command line) as parameters.
- The Server and UserNodes must use ProjectLib’s send and receive methods to communicate. You must not use any other communication methods (e.g., Java RMI or sockets). You should not use the filesystem as a means to communicate or to perform actions on behalf of other entities.
- Your code should handle lost messages. You can assume that a message will take no more than 3 seconds to be delivered if it is not lost.
- A new candidate collage will be posted to your Server using the startCommit method. The arguments to this will provide the filename and contents of the collage, and an array of strings indicating the source images. Each string is of the form “address:filename”, where address is the id of the UserNode that owns the image, which has the given filename at that UserNode.
- A UserNode can ask the “user” if a candidate collage is acceptable using the askUser method of ProjectLib. If the return value is true, then the user is happy with the collage.
- A UserNode should ensure that each of its images can appear in no more than one committed collage.
- If a collage is committed, then it should be written to the Server’s working directory with the given filename. All UserNodes should remove the source images that contributed to the committed collage from their working directories.

- Messages can be lost at any time, and any UserNode or the Server can be killed and restarted at any time. The network may fail. These failures may be asymmetric. Your code should be robust to these failures.

### Submission and grading:

- You will be graded on the correctness of your system. The test cases will verify that the right set of collages are added to the Server working directory, and that the corresponding source images are removed from the UserNode working directories.
- Although performance is not critical, you should limit the number of messages sent. Too many messages will fail the test cases as well.
- This project will use an autograder to test your code. See below on how to submit.
- You need to submit a short (1-2 pages) document detailing your design. See below.
- The late policy will be as specified on the course website, and will apply to the checkpoints as well as the final submission
- Coding style should follow the guidelines specified on the course website

### Checkpoint 1 (40%) Due: 11:59 PM, Tuesday April 15, 2025

Checkpoint 1 requires you to implement a two-phase commit to publish the collages. Your code should be able to handle both successful and unsuccessful commit operations. It should allow multiple candidate collages to be processed in parallel. Committed collages should be written to the Server's working directory, and any images used in committed collages should be removed from the UserNode directories. Node failures or message failures will not be tested.

### Final (60%) Due: 11:59 PM, Tuesday April 22, 2025

The final submission needs to handle failures. It should be able to handle the loss of any messages. The Server or any of the UserNodes can be killed and restarted at any time. These components should recover in the manner described in lectures and this writeup. (40%)

You will also need to write and submit a 1-2 page document, describing the major design aspects of your project, including your protocol between Server and UserNodes, timeout thresholds, how you handle lost messages, and how you recover from node failures. Highlight any other design decisions you would like us to be aware of. Please include this as a PDF in your final tarball. (10%)

Your final source code will also be graded on clarity and style. (10%)

## Submission Process and Autograding

We will be using the Autolab system to evaluate your code. Please adhere to the following guidelines to make sure your code is compatible with the autograding system.

First, untar the provided project 4 handout into a private directory not readable by anyone else (e.g., ~/private in your AFS space):

```
    cd ~/private; tar xvzf ~/15440-p4.tgz
```
This will create a 15440-p4 folder with needed libraries, classes, and test tools. You should create your working directory in the 15440-p4 directory. It is important that from your working directory, the provided java classes should be available at ../lib.

Write your code and Makefile in your working directory. You must use a makefile to build your project. See the included sample code for an example. You will need to add the absolute path of your working directory and the absolute path of the lib directory to the CLASSPATH environment variable, e.g., from your working directory:
```
    export CLASSPATH=$PWD:$PWD/../lib
```
Ensure that by simply running "make" in your working directory, your Server, UserNode, and support classes are built. Please use the names "Server" and “UserNode” for the two required classes. Make sure the java and generated .class files are in your working directory (i.e., not in a subdirectory). Your Server and UserNode classes should implement main. Do not place your classes in a java package! Leave them in the default package. This naming convention and relative file locations are critical for the grading system to build and run your programs.

To hand in your code, **from your working directory**, create a gzipped tar file that contains your makefile and sources. E.g.,
```
   tar cvzf ../mysolution.tgz Makefile Server.java UserNode.java ...
```
Of course, replace these with your actual files, and add everything you need to compile your code. If you use subdirectories and/or multiple sources, add these. Do not add any files generated during compilation (e.g. the .class files) -- just the clean sources. Also, do not add the class files that we have provided -- these will be installed automatically when grading. To work correctly with Autolab, when extracted, your tarball should put the Makefile and sources in the current working directory.

You can then log in to **https://autolab.andrew.cmu.edu** using your Andrew credentials. Submit your tarball (`mysolution.tgz` in the example above) to the autolab site. Note that each checkpoint shows up as a separate assessment on the Autolab course page. For your final submission, include your write up as a PDF file in your tarball.

## Recovering From Failures and Lost Messages

The lecture material on two-phase commits shows the simplest algorithm that works with a (possibly unreliable) datagram messaging protocol like the one used in this project. Note that in the lecture notes, retransmission is avoided as much as possible. If the prepare message or a node reply is lost, then the coordinator just assumes the vote is "no" after a timeout. This avoids retransmissions of the prepare or vote messages. However, after the commit/abort point, the information MUST eventually propagate to the participant nodes. So here, the server does need to keep sending out the result until all participants have replied with an ack. (Alternatively, the other nodes can keep asking the server node until they get the result of the transaction).

Likewise, please keep your recovery code as simple as possible. Recall from the lecture notes on Write-ahead logging that recovery is the most dangerous time for a server, and cascading failures (i.e., failures during recovery) do happen. So it is much more important to have very simple code that is much more likely to be bug-free than to try to do fancy processing during recovery. Thus, it is best practice to simply abort all non-committed transactions during recovery. It is not worth the complexity to try to resume transactions that have not committed, especially since failure should be a very rare event. However, on recovery, your server cannot forget / abort transactions that had successfully reached the commit point. Likewise, a node that votes "yes" cannot forget this vote / change its mind even if it crashes and recovers.

We expect that you will structure your implementation and handle lost messages / failed nodes based on these principles from the lecture notes.

## How to Use the Supplied Classes and Files

### ProjectLib Class

We will provide a class called ProjectLib. This is the main library that you will use for this project. It provides several methods that emulate OS functions that are needed for communications and performing transactions locally.

Communications are based on a unidirectional, unreliable datagram service, conceptually similar to UDP. ProjectLib.Message is a simple message class that includes an address (String) and a message body (byte array). When sending a message, the address should be set to the destination node's id; when receiving, it contains the sender's node id. ProjectLib provides a sendMessage() method for sending a message to another node. No information about success / failure is provided by this method. A getMessage() method is a blocking call to pull the next message out the node's receive queue. The receive queue is FIFO, but since messages can be delayed or dropped before being put in the queue, order of receipt is not guaranteed. Messages are never corrupted -- only delayed or lost.

Instead of using the receive queue and the blocking getMessage call, your code can use a callback mechanism to receive messages. To do this, you need a class that implements the ProjectLib.Messagehandling interface, which defines the callback function prototype. To register the callback, simply supply a reference to this class as an additional parameter to the ProjectLib constructor (see below).

Since your code needs to handle failures / reboots of the Server and UserNode, you will need to keep some state in persistent storage. This can be a bit tricky because a crash may happen at any time and may affect either the application or the system as a whole. To ensure data really is written to disk, your code must first flush application-level buffers or close files -- this will ensure the data makes it to the OS. But the data may still be in OS memory (disk cache). In a real system, you should then call () or the equivalent to ensure data is flushed from OS buffers to the disk. Since system failures and reboots are emulated here, you should use the fsync() method provided by ProjectLib rather than the OS version. On an emulated crash/reboot, when your code is restarted, it will see files exactly as they were at the last call to ProjectLib.fsync(). (Note: this is a simplification;  in real systems, the disk state is guaranteed to have all writes before the last successful fsync(), but may have some subset of the later writes).

Both your Server class and UserNode class should instantiate ProjectLib.  The UserNode should supply the constructor with <id> and <port>, as provided on the command line.  This <id> will be the “address” of the UserNode for messaging.  The Server needs to provide a callback function for starting the commit process, as defined by the ProjectLib.CommitServing interface.  The Server should provide the <port> (from the command line), and a reference to a class instance that implements the CommitServing interface. The Server will be assigned an address of “Server” for messaging.  Optionally, when instantiating ProjectLib, both the Server and UserNode can supply an extra parameter -- a reference to a class instance that implements MessageHandling -- if it wishes to receive messages using a callback function.

## Project4 class

The Project4 class is used to start the test environment, launch your Server and UserNodes, and run through a test scenario.  To run the program, ensure your CLASSPATH is set correctly (and includes absolute paths to both the lib directory and the directory with your compiled classes), then execute:

```
      java Project4 <port> <script_file>
```

Here, <port> specifies the port that it will use internally to communicate between components. The <script_file> contains a set of commands used in a particular test, indicating the set of UserNodes to start, candidate images to commit, which messages are delayed or dropped, and which nodes if any fail or are restarted.  Project4 will launch your Server and UserNodes as separate processes, running in separate working directories.  It expects these directories have already been created, and pre-populated with the image files at each node.  Project4 will produce a unified stdout/stderr from all of the processes launched.

## Test.tar

We provide a set of test images and scripts in test.tar.  Please untar this to create a directory called test.  This will contain a set of subdirectories that will contain images and will be the working directories for your UserNodes.  There will be an empty directory for the Server.  In addition, a set of candidate collage images will be provided, along with a set of scripts for running Project4.  To run one of the scripts, cd into the test directory, and run Project4 (see above), supplying the appropriate script file on the command line.  Running Project4 will clutter the test directory -- your Server and UserNodes will create and delete files, and the fsync() operation will create backup copies of file state.  To run another test, we recommend that you recursively delete the entire test directory, and extract a clean copy from test.tar.

## Custom Script files

We provide a few example scripts in Test.tar to get you started.  We encourage you to construct your own scenarios and write corresponding script files for testing your system.  The script file is a simple text file that lists a sequence of commands that the Project4 class will execute.  The class uses a global notion of time (specified in milliseconds).  Lines starting with ‘#’ are comments and are ignored. Likewise, blank lines are ignored. The test ends when the end of the script file is reached. All other lines must be one of the following commands:

- `start <id1> <id2> …`
  Start / restart the node(s) named in the list. You should have one node named "Server", which will run your Server code. Other nodes can be named anything. These will run your UserNode code. The nodes will run in subdirectories of the same name. When restarting a node, the state of the directory will be set to what it was when the last call to fsync was made.
- `kill <id1> <id2> …`
  Immediately kills the node(s) in the list.
- `restart <id1> <id2> …`
  Equivalent to kill followed by start.
- `commit <collage> <src1> <src2> …`
  Tells the Server to start a 2PC for a new collage. <collage> is of the form "name=path" or "path". Name is optional, and indicates the filename of the collage output image that will be written to the server's directory on successful commit. Path is the actual image corresponding to the collage. <src1>, <src2>, etc. are the list of the contributing source images. Each is of the form "id:filename", indicating the node id and the name of the image file in that node's directory.
- `setDelay <id1> <id2> <delay>`
  sets the one-way message delay from to <id1> to <id2> be <delay> milliseconds. Note that this is not symmetric – if you want both directions to be changed, you need to set each direction separately. The node ids can also be "*", which indicates any node. E.g., to set the default delay (i.e., any node to any node) to 100 ms, use:
  ```
      setDelay * * 100
  ```
  To indicate messages are to be dropped, set <delay> to -1. E.g., to start dropping messages from B to the Server, use:
  ```
      setDelay B Server -1
  ```
  These commands take effect immediately and are applied to any matching messages sent afterwards (not those in flight).
- `wait <delay>`
  Pause the script for <delay> milliseconds, allowing the server and usernodes to run / make progress. Without the addition of waits, commands are executed back to back with no delay (effectively, they happen all at once). You should put wait commands after start commands to give nodes a chance to start running. Likewise, remember to put a wait at the end of the script to let execution complete. At the end of the script, Project4 terminates, immediately killing all of the nodes as well.

It is recommended that you set a default message delay right at the beginning of your script. Remember to add waits between commands that should be spaced in time, and to add a wait at the end to allow execution to complete.

## Notes / Hints

- You should not use any form of communication between your nodes except through the ProjectLib messaging services. This means no RMI, no sockets, no sharing files or peeking into the working directories of other processes, etc.
- There is only one receive queue for messages in each node. So messages dealing with concurrent commit operations will be mixed together. You may want to add your own layer of messaging APIs on top of ProjectLib's services, perhaps extending them to provide separate receive queues for each concurrent commit operation.
- You can use the asynchronous callback mechanism and the blocking getMessage() to receive messages at the same time. When a message arrives, your callback will be called. If it does not want / know what to do with the message, it can return false, and the message will be put into the receive queue, where the blocking getMessage method can retrieve it.
- Recall that two-phase commit does not provide guarantees on termination. To avoid blocking forever, you will need to implement timeouts for various parts of your two-phase commits. You will need to determine reasonable timeout intervals for your system. You can assume that a message that is not lost is guaranteed to arrive within 3 seconds of being sent.
- This project requires serializing data when you write to disk and transmit messages across the network. You are welcome to serialize data any way that works. You may wish to look into Java's ObjectOutputStream class, FileOutputStream, ByteArrayOutputStream, etc.
- Conceptually, you will use local transactions in building the distributed two-phase commit transaction. However, you may not need to implement a complete shadowing or intentions list / write-ahead logging local transactions system to get this project working. Your solution does need to implement atomicity and durability as a local transactions system would provide.
- Because your code is expected to write and delete files to a set of directories, it is recommended that you extract a fresh copy of the test.tar file for each test run.