SubMIT:

Redesigned Submission and Grading System for
6.033: Computer System Engineering
at the Massachusetts Institute of Technology

Jeremy Cowham, Christian Moroney, and Russell Pasetes
MIT Department of Electrical Engineering and Computer Science
{ jcowham, cmoroney, rpasetes } @mit.edu

Recitation Instructor: Mohammad Alizadeh
WRAP Instructor: Juergen Schoenstein
TA: Steven Okada
Time: TR 1PM
May 6, 2019

Table of Contents

Introduction

The MIT class *Computer System Engineering* (6.033) currently utilizes a range of systems for handling files, submissions, and grades in a way which is problematic for students and staff alike. These issues include but are not limited to: split systems for handling grades, poor implementation of submission sites, and the lack of a platform supporting shared group work. These problems have plagued the 6.033 community for too long, and a new system needs to be designed. Focused on solving these problems, we design a new system for 6.033, named “SubMIT”, that leverages existing systems, but reengineers the implementations and interactions in a way which best meets the needs of students and staff.

A central server that communicates with all students and staff provides the foundation for our system. In designing SubMIT, we identified the core design principles of *simplicity, security,* and *utility*. Simplicity of the system means, for example, enabling students from different academic backgrounds to adapt quickly to the logistics of the class. We emphasize the intuitiveness of the storage and file management system, promoting safe and effective work by students and staff alike. Furthermore, the security of SubMIT means avoiding technical failures, ensuring storage safety, and thwarting malicious activity. Finally, the utility of SubMIT means, in essence, addressing significant user needs. This principle informs design decisions so that trade-offs and compromises are settled in a way which achieves simple, user-optimal solutions. SubMIT facilitates the communication between and among users and infrastructure services to meet the needs of students and staff.

System Overview

Our system utilizes the infrastructure services at our disposal: the MIT ID service (MIDS) based on the Kerberos system, the MIT File System (MFS), the MIT Sync Service (MSS), the MIT Locking Service (MLS), and Gradescope. Our system establishes a multipurpose platform that supports graded assignments for students in 6.033. Every communication request made by a user, either staff or student, will first be interfaced through a submission website that authenticates requests through MIDS in order to coordinate with SubMIT and sets proper access parameters. This enforces a secure framework for student and staff interactions. Within the server, we store all student data sent to SubMIT in the file system hierarchy. We delegate access control to concurrently shared files between Design Project teams through MLS. Additionally, we update our databases with information as students submit assignments. The databases are only accessible by staff, and handle many of the features staff needs in order to run the class. Staff possesses the ability to share data from the databases with student’s individual folders for assignment feedback.

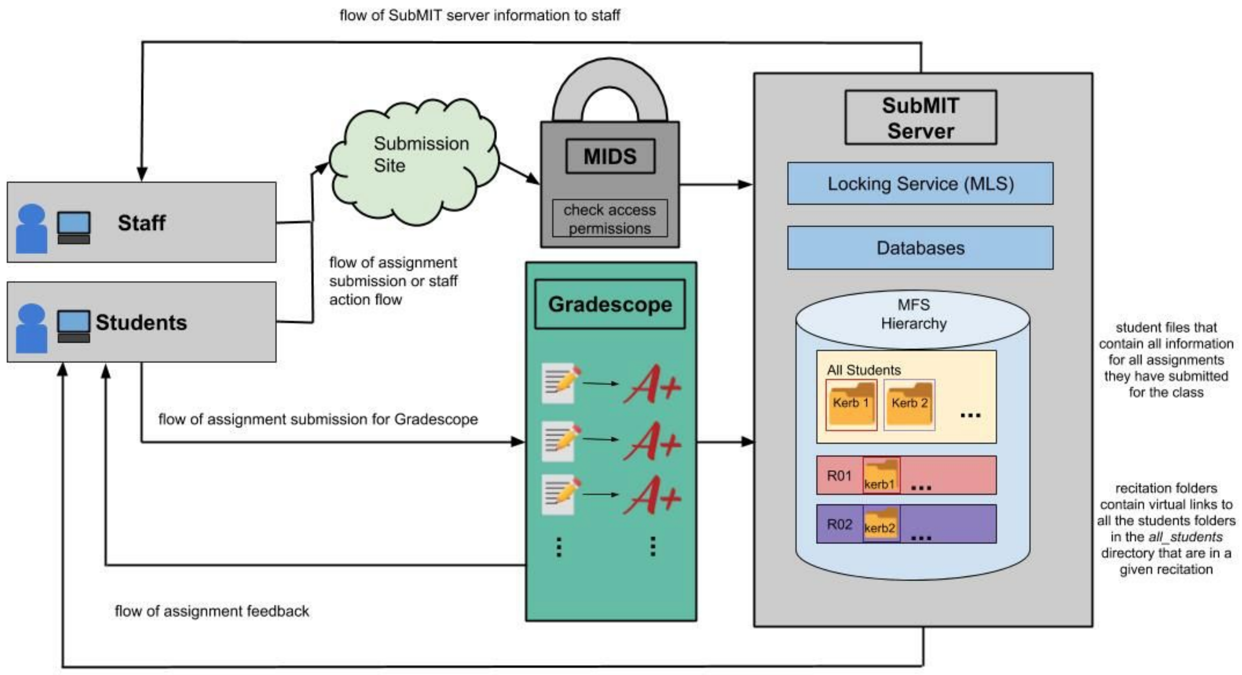

flow of SubMIT server information to staff
Submission Site
MIDS
check access permissions
SubMIT Server
Locking Service (MLS)
Databases
Staff
Students
flow of assignment submission or staff action flow
Gradescope
MFS Hierarchy
All Students
Kerb 1
Kerb 2
...
R01
kerb1
R02
kerb2
student files that contain all information for all assignments they have submitted for the class
recitation folders contain virtual links to all the students folders in the all_students directory that are in a given recitation
flow of assignment submission for Gradescope
flow of assignment feedback

Figure 1: *The main interactions of our system with users. Students and staff access our system via remote computers. MIDS validates user access for all requests. MLS dictates who can read or write to files in the MFS hierarchy. Gradescope periodically will report grades to the SubMIT server. Students will upload an assignment to either Gradescope or SubMIT and receive feedback for the assignment on the respective system.*

File System

The main goal in design of the SubMIT file system was to design an understandable hierarchy that was simple for students, a course lecturer, and various recitation instructors or teaching assistants to interact with. With these design goals in mind, we structure our file system in a way that has an *all_students* directory along with a *recitation_X* directory for all recitation sections for the given class. This structure provides the foundation for the folders that students will interact with on the submission site. Displayed below (Figure 2) are the various *views* different clients of the system will use. We limit students to viewing their own specific kerberos files along with their design project team folders. Additionally, we limit staff to only viewing the recitation directories with which they are associated, reserving total access to the file hierarchy for the course lecturer. We provide the course lecturer additional space in the root directory to store metadata, flags of data issues in the SubMIT system, such as when communicating with Gradescope, and additional information that the lecturer deems necessary to store. This design choice ensures simplicity as clients only interact with data that they should be concerned with, along with a layer of security.

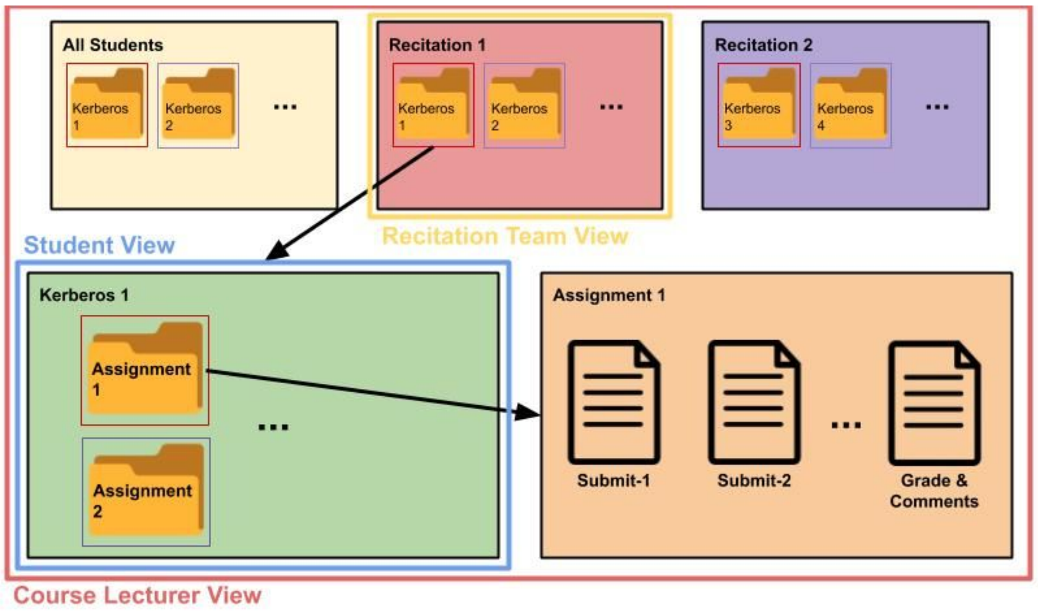


Figure 2: *This figure displays the directories found in our root directory of our SubMIT server's file system along with the scopes of the various users of our server: course lecturer/staff, recitation teams, and students. Note: recitation teams and students' directories take the same format.*

Databases

SQL databases are an ideal data structure for storing repeated, similarly-formatted information and metadata. There are several components of 6.033 which match this criterion including assignments, submissions, grades, and video voting. SubMIT, therefore, uses three databases: an assignments definition database, a combined submission and grades database, and a video voting database.

The assignments database contains a record of all course assignments and stores the following fields for each entry: assignment ID, link to assignment description, assigned to boolean (0 for individual, 1 for team assignment), due date/time, grading permissions for staff, percentage of final grade, late policy, submission required boolean (some assignments, like recitation grades and DP presentations, do not require a submission), and Gradescope boolean (whether the assignment is to be graded on Gradescope or SubMIT). This database is required to be formed prior to the start of the semester, as when bulk creation of accounts and folders occurs, this database will serve as the template for the Kerberos-specific folders. We assume the course lecturer will have the knowledge prior to the start of the term regarding what assignments will be included throughout the semester. While assignments may be updated, configuring assignments before the start of semester provides convenience for staff.

In order to maintain the link between submissions and grades, these components are unified into one database. To accomplish this, each assignment for each student will have a database entry, even assignments with no submission. SubMIT maintains the following fields for entries in this database: assignment ID (identical to the corresponding assignment database entry), submitter Kerberos, date/time of submission, address of the submission in the file system (null if on Gradescope), the grade (null until entered), an optional comment field, *published* boolean (0, or private, by default), a multiplier for late penalty (1 by default), and a Gradescope boolean. We store both grade and optional comment fields as lists, as this supports the concurrent uploading of feedback from multiple staff members at the same time. A recitation instructor can append their grade and comments at the same time as a WRAP instructor. If one instructor publishes feedback before the other is able to upload their feedback, we automatically update the file system with all grades and feedback for assignments where the *published* boolean is 1.

Recording votes on video submissions will be done through a database specially for this purpose. Entries in this database will maintain the following fields: submitting team Kerberos, address of the video in the file system, and a tally of votes received for each respective rank (from 1-5). The tally of votes is acquired through the parsing of assignment completion that comes from the students that rank the videos.

One final component of SubMIT databases is a “completed” bit appended at the end of each entry of each database. Zero by default, this bit ensures atomicity of data transfers, and is discussed in detail in the data transfer protocol.

The choice to include these databases was to build further on our design goal of simplicity of user interactions. We believe that each of these databases serve a unique purpose to the overall functionality of our system.

Student Interactions

Assignment Organization & Access

Only the Course Lecturer, head WRAP Instructor, and Administrative TA have read and write access permissions for the *all_students* directory because the staff may need to add or remove students as registration changes. This folder is hidden from all students to ensure that a student cannot access another classmate’s directory.

Each recitation subdirectory gives read and write access permissions only to the corresponding Recitation Instructor, Recitation TA, and WRAP Instructors. Within each recitation subdirectory are symbolic links to the student directories in *all_students* within that recitation section. Each student has read and write access to their own student directory along with the directory created once design project teams are formed.

Once teams are formed, a team directory is created using *create_dir*(kerb1,kerb2,kerb3,kerb4=null) for all Design Project (DP) teams within the recitation section. These team directories can only be created if and only if all team members’ individual directories exist within the same recitation section. Furthermore, these team directories exist in the recitation section subdirectory that the students are assigned to. All team members within a DP team are given read and write access to their corresponding team directory. We also place a lower limit on the minimum number of students in a given team. This ensures that there will be multiple teams of three students, with a few edge cases of four students on a DP for the overall class.

We design SubMIT's assignment storage in this way to support overall simplicity of student interactions. We believe that storing assignments in this way is logical and understandable, which avoids students finding themselves confused and decreases likelihood of client-side errors. Not only is this structure beneficial for simplicity, but also security. With the proper access control, we avoid students obtaining files or other data that they are not supposed to be possessing.

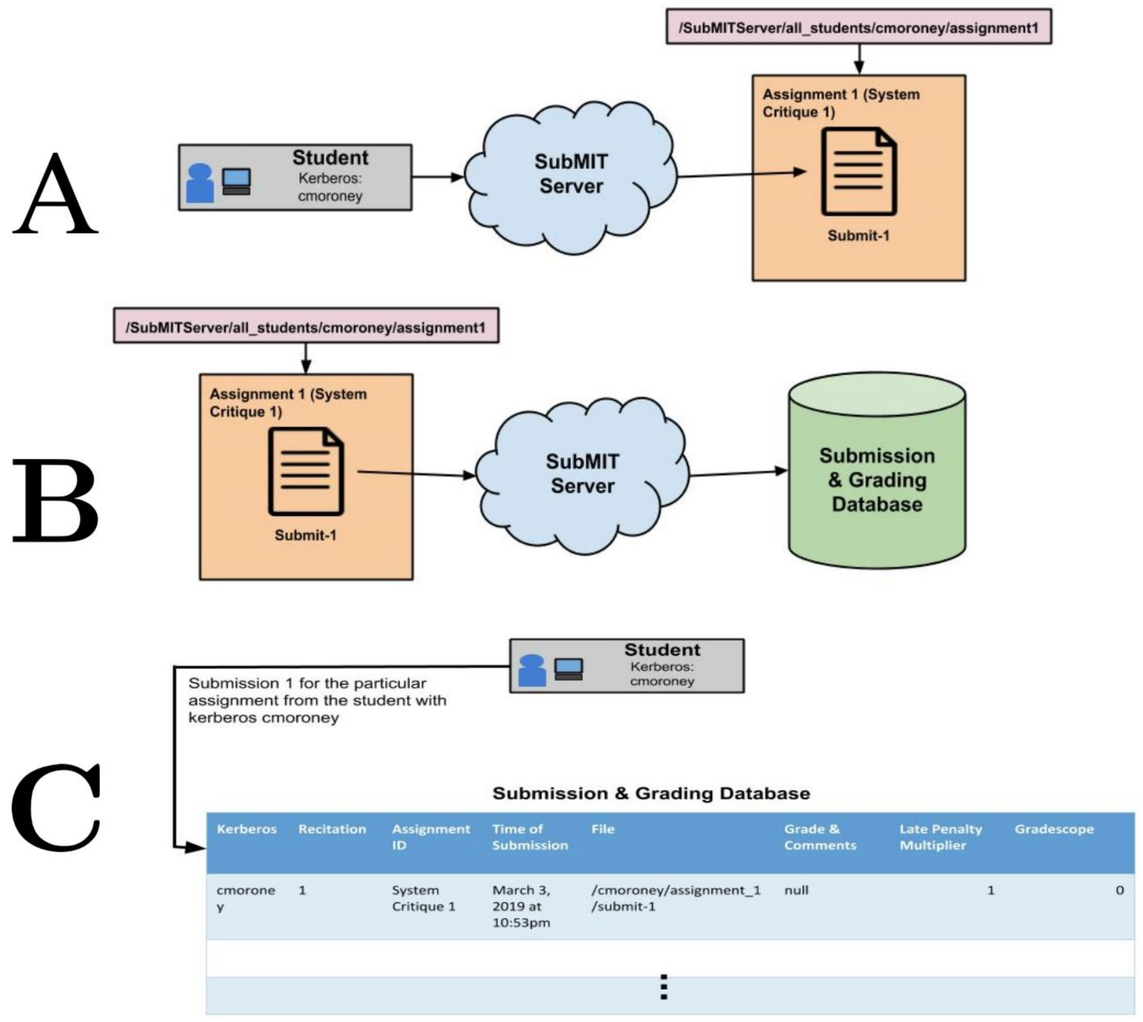
Assignment Storage

Inside each student directory, subdirectories are created that hold each individual grading element for the class. Each individual assignment subdirectory is created based on the entries within the assignment database at the start of the semester, which is only write accessible by the Course Lecturer. For the purposes of 6.033, there exists three subdirectories: reading questions, critiques, and peer reviews. These subdirectories allow for the upload of multiple files if a student decides to submit an assignment more than once. Inside each team directory, subdirectories are created that hold each team grading element for the class: the Design Project Preliminary Report (DPPR), the Design Project Report (DPR), and the Poster Video. Similarly, these subdirectories allow for multiple file uploads if a team decides to submit an assignment more than once, with the exception of the Poster Video. In the event of multiple Poster Video uploads, we overwrite previous submissions. We choose to design SubMIT in this way to avoid large amounts of memory being allocated to video storage, where staff will only be checking the most recent submission. In these student/team directories, we allocate space for students to work on a editable files, which can be ultimately submitted via the subMIT server, but saved as progress is made. SubMIT does not keep a fully history of these files, but rather stores one copy. However, SubMIT does store a full history of submissions to the server for a given assignment.

Submission Function

In order to support the submission of assignments, we provide students and DP teams with the function *submit_assignment(*filename, assignment_ID), which gives students the ability to set a file as a submission for an assignment to the given directory if they hold the proper Kerberos ID permissions. A copy of the file is created within the specified directory and is tagged as the most recent submission with a timestamp of when this function was called. Any previously uploaded file will retain their timestamp tags but only the tag with the most recent timestamp will be recognized as the submission considered for grading.

Once the file is copied in the directory, *submit_assignment* will populate the corresponding assignment ID entry that's tied to the matching Kerberos ID within our submission and grading database. The date/time of submission will be filled by the submission timestamp and the address of the submission will be a pointer to the assignment within our file system. Once these fields are populated, the submission is ready for grading by staff. If the database entry is already populated, only the date/time of the submission and address of the submission will be updated with the most recently uploaded submission.


A
/SubMITServer/all_students/cmoroney/assignment1
Student
Kerberos: cmoroney
SubMIT Server
Assignment 1 (System Critique 1)
Submit-1
B
/SubMITServer/all_students/cmoroney/assignment1
Assignment 1 (System Critique 1)
Submit-1
SubMIT Server
Submission & Grading Database
C
Student
Kerberos: cmoroney
Submission 1 for the particular assignment from the student with kerberos cmoroney
Submission & Grading Database

Kerberos	Recitation	Assignment ID	Time of Submission	File	Grade & Comments	Late Penalty Multiplier	Gradescope
cmoroney	1	System Critique 1	March 3, 2019 at 10:53pm	/cmoroney/assignment_1/submit-1	null	1	0
			⋮				

Figure 3: ***(A)*** *As a student with kerberos cmoroney submits a file for Assignment 1 (System Critique 1) we populate the cmoroney subdirectory labeled Assignment 1 with a copy of the submission file.* ***(B)*** *the process of submitting this file starts the process within the SubMIT Server where SubMIT updates the submission and grading Database with the appropriate entry.* ***(C)*** *At a high level, the process of submitting assignment 1 from kerberos cmoroney causes the updating of the specific entry in the database. Note: students do not know that they are interacting with this database, the updating of the database is handled completely within SubMIT.*

Team File Sharing

Group projects constitute a significant component of the 6.033 curriculum, so SubMIT implements a team file sharing mechanism which strikes a balance between real-time concurrent editing and exclusive editing ability.

SubMIT designates files as either concurrently editable (CE) or non-concurrently editable (NCE). CE files allow team members to check out portions of a shared text file by section. To accomplish this, CE files must follow strict formatting guidelines: each section available for check-out must have a unique section title (e.g. "Introduction", "Student Interactions"). These unique titles must be present in a table of contents and be an *exact* match. These titles will be used to delineate each editable section. A section includes its title and any text up to but not including the title of the next section. NCE files are much more versatile, as they do not need to adhere to a strict format; however, as the name implies, only one user may edit an NCE file at a time. NCE files can include PDFs, images files, and even text files designated as such.

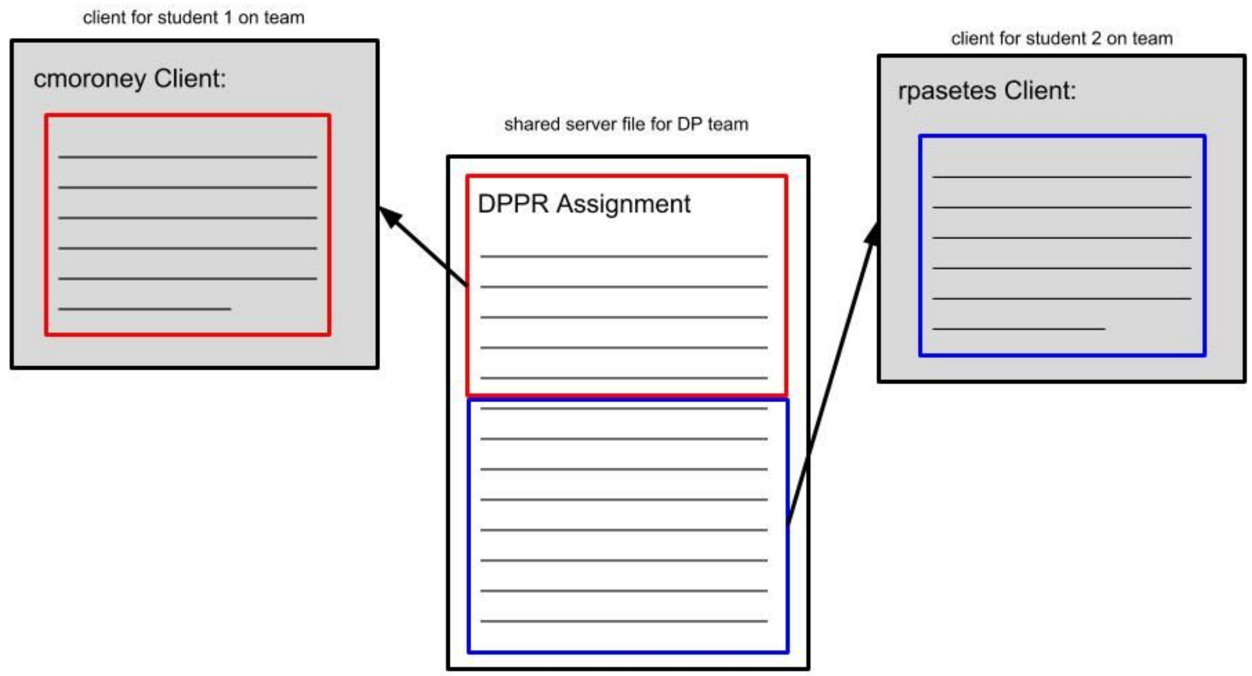

client for student 1 on team
cmoroney Client:
shared server file for DP team
DPPR Assignment
client for student 2 on team
rpasetes Client:

Figure 4: *The shared file in the diagram will exist in a DP team's subdirectory for the specific assignment. Student 1 with kerberos, cmoroney, can select portions of the document to 'check-out' and work on. This prevents any other student, such as student 2 with kerberos rpasetes, with write access to the portions of this document already checked out. The respective red and blue sections of the assignment illustrate the sections currently 'checked-out' and the work capable of being done on by separate clients for the same file on separate devices.*

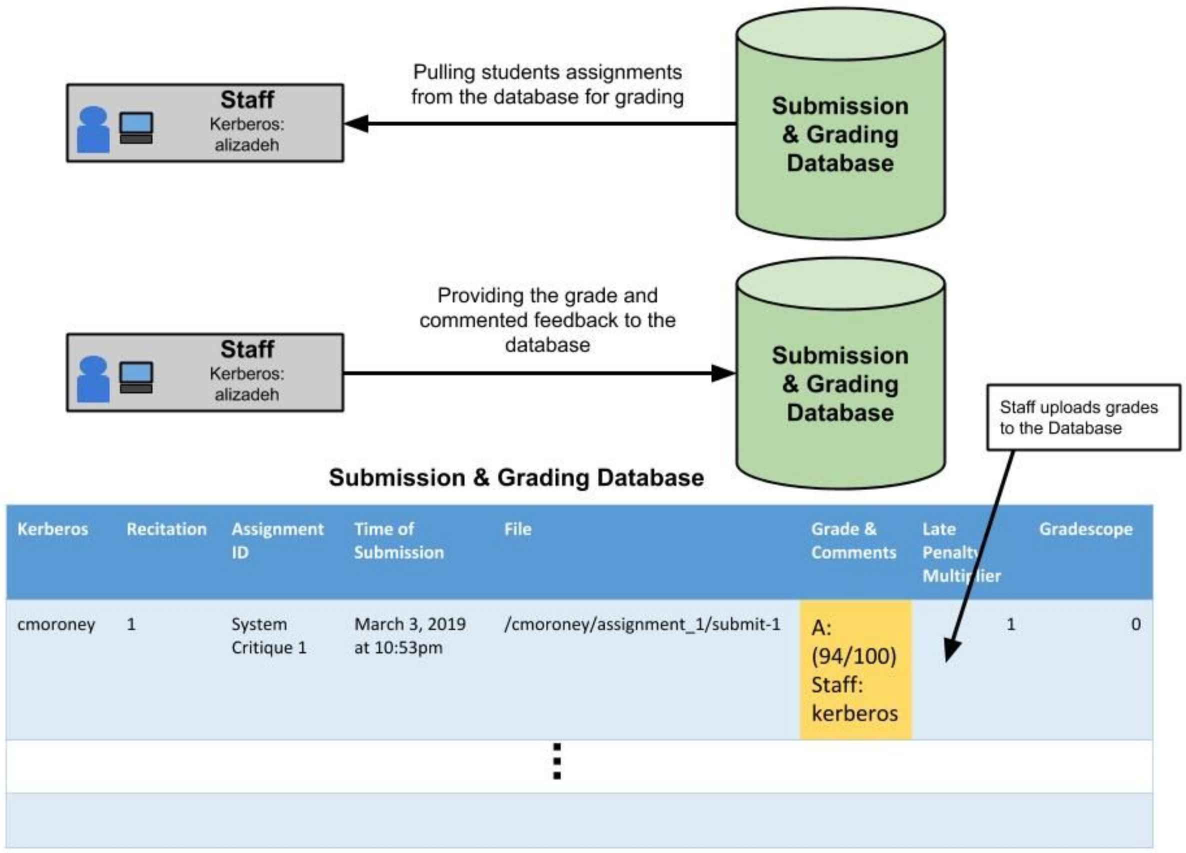
We handle the complex challenge of CE files by identifying sections in the file based on the bytes in which the section starts and stops through identifying sections in the table of contents. In a metadata folder in an assignment-specific folder, which is in the design team's personal directory, we store these byte values to keep track of sections along with a mapping of the section title to the byte value associated with the start of the section. When a team member wants to work on a given section of the file, they *checkout* that section by acquiring a lock on this section. This member solely possesses the ability to edit this section, until it is checked back into the system, at which point the lock is released. This ensures that multiple members can be working on different sections at once.

While a section is checked out, other team members have access to view the most recently checked in version of the section. This version control saves all check ins for every section of a file in the assignment directory. In the event that a section has changed in overall size, in other words, the difference in bytes between the start and end have increased or decreased, the section that is sequentially after it on the file will have its new start and end byte values updated in the metadata. SubMIT continues the process to shift all start and end bytes that are affected by the change.

This team file sharing scheme aims to do away with the resource-intensive processes and costly application design for a service like Google Docs, but also provides teams with the ability to concurrently edit the most common type of 6.033 group assignment: text documents. We design SubMIT in this way to achieve a more simple process of team collaboration on assignments. As alumni of 6.033, we design with this approach based on our own experience with team assignments. We recognized that essentially all assignments submitted as a team are files taking on the form similar to a research paper, or thesis. With multiple sections of the paper, our approach to completing the final deliverable was divvying up sections to write up individually, and merge these sections together for the final product that would be submitted. We acknowledge the design tradeoff of slight overhead, and increased backend complexity to achieve a more simple and understandable interface for design teams to collaborate with.

Staff Interactions

Grades & Comments


Staff
Kerberos: alizadeh
Pulling students assignments from the database for grading
Submission & Grading Database
Staff
Kerberos: alizadeh
Providing the grade and commented feedback to the database
Submission & Grading Database
Staff uploads grades to the Database
Submission & Grading Database

Kerberos	Recitation	Assignment ID	Time of Submission	File	Grade & Comments	Late Penalty Multiplier	Gradescope
cmoroney	1	System Critique 1	March 3, 2019 at 10:53pm	/cmoroney/assignment_1/submit-1	A: (94/100) Staff: kerberos	1	0
⋮							

Figure 5: *The logic behind our database for submission and grading. Staff can pull the file that a student submitted for an assignment, grade that assignment, and then upload the grade to the Database. This field in the database is also able to store comments that staff has the option of filling in.*

With the student assignments stored on our SubMIT server and Course Staff able to access students files with read or write privileges, we provide staff the capability to supply feedback to students. Within the file system on the SubMIT server, we allocate space for a database that will be accessed by relevant staff to provide feedback to students for assignments. This feedback can take the form of either a strict letter grade or a grade and some comments. Relevant staff for grading of an assignment will access the database and query for the recitation section that they are associated with and for the particular assignment they are grading. Staff members then have access to all the files that need feedback. Once the staff member has determined the grade and comments, they can upload these to the SubMIT server. This process strictly updates the database with the information provided by staff. Furthermore, this uploading of data tags it with the kerberos of the grader and stores it in both the grade and comment sections. The publication of these grades is handled separately.

Late Penalty

Within the metadata of each uploaded file is a timestamp of the time of submission, which will be automatically checked against the due date of the given assignment and will apply the appropriate penalty based on the policy explicitly stated on each assignment (both accessible by the assignment database). This is done by adjusting the late penalty multiplier in the submission and grading database. Administrative staff have the option to waive the penalty applied to any given assignment, based on extenuating circumstances (eg. note from an S3 dean). To provide slight leeway to students, late penalties will not be enforced until 10 minutes after a deadline to alleviate stress of network traffic (large portion of submissions right at the deadline).

Grade Publication

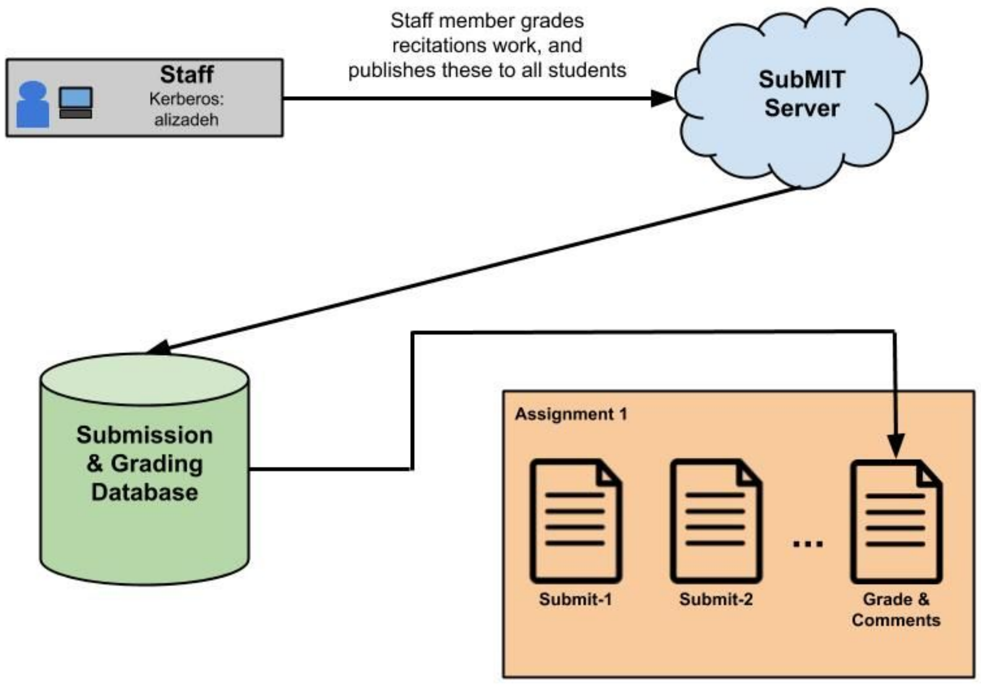

Staff member grades recitations work, and publishes these to all students
Staff
Kerberos: alizadeh
SubMIT Server
Submission & Grading Database
Assignment 1
Submit-1
Submit-2
...
Grade & Comments

Figure 6: *Staff utilizes the publication function to send the data stored in the Grade and Comment section of the Submission & Grading Database into each student's specific assignment folder. This gives students access to view the grade feedback.*

Once staff has completed inputting grades and comments into the database, staff utilizes the *publication* function on the SubMIT Server to update student's specific assignment folders with a file that stores information regarding the grade feedback. This file includes both the grade received and the comments for the assignment if staff provided any. Furthermore, this stores the information about who provided the feedback for that assignment. We provide the *publication* function only once staff has graded an entire recitation's submission for an assignment. We design the system in this way to ensure that staff is not burdened with the task of manually publishing every grade for each individual student, as this repetition might lead to human error on the staff side. Furthermore, this design choice emphasizes the simplicity of our system, as staff only has to worry about publishing on a per assignment basis for their given recitation.

Gradescope Integration

As our system is the primary source for grade feedback for 6.033, we design SubMIT in a way that it interacts with Gradescope to maintain an eventually consistent report of Gradescope grades. As some assignments for the course require submittal via Gradescope, SubMIT communicates with Gradescope to update student's grades on the SubMIT Server. For an assignment that is handled in Gradescope, we store a directory on the SubMIT Server in every student's personal directory for that assignment. In times where activity from clients of SubMIT is expected be low, specifically in the morning at 4am, we pull a snapshot of the current grades on Gradescope to update the SubMIT file system with. We then iterate through the snapshot and upload grade information into each students kerberos-specific folder and upload a file containing grade information into assignment-specific folders. We perform this in times of low activity to avoid jamming the network with traffic for an operation that we hold to lower priority than student or staff data uploads. In the event that there is a grade for a student that is not identified in the *all_students_directory* we flag this attempted entry, and notify the course lecturer. The course lecturer can determine what the error was, and take necessary actions to handle the situation. However, in the event that a student drops the course, we store this student's kerberos in a file stored in a personal directory in the root for the course lecturer to view. In scenarios where Gradescope attempts to sync information and a kerberos is not found, we do not flag failed entries if the kerberos exists in the file of dropped students. We design the system in this way because we determine that these errors happen rarely, and can be handled by the head lecturer in the event that they occur.

We assume students do not need to see Gradescope grades on the SubMIT server immediately as they are posted on Gradescope, which justifies our idea of eventual consistency between the two systems. Furthermore, staff will have a solid idea of the time it takes to grade assignments, and thus our system will perform these pulls of Gradescope data every week after the due date for these assignments. We also assume that staff can set deadlines for regrade requests, and thus can perform another pull a week after regrade deadlines to update any information that may have changed.

Grade Collection

The Submissions & Grading database serves a great function of separating staff interactions with grades from student interactions with grades. The database allows for concurrent addition of grades from multiple staff members, and the multiple fields of specificity allow for the querying of grade reports. Staff members will be able to clarify what recitation or assignment they wish to see all the grades for. The database also serves the function of being the template for the end-of-term spreadsheet for the grades meeting. We assume that there will be virtually no activity on the server at the end of the term, as students will have all assignments submitted. This allows for the SubMIT server to have the processing allocation necessary to support this feature.

Server Processing

Storage Resource Allocation

The SubMIT server boasts 240 GB of duplicated disk storage which must be allocated prudently. There also exists another 16 GB of main memory; however, this storage is discussed in the next section as it relates to short-term memory usage.

There are three main categories of data SubMIT will maintain in long-term memory: OS kernel code, student assignment submissions, and staff databases. As we modeled SubMIT after the Unix file system, Linux is a viable choice of software for the server. Installation of Linux typically requires 4 - 8 GB of space[1], so 8 GB is a reasonable estimate for space occupied by the OS kernel. For written student submissions, the calculation becomes a bit more intricate. According to details from the DP description, there are approximately 40 assignments per semester, 36 of which are submitted individually and four of which are submitted as a team. In 6.033, individual assignments tend to take a more strictly text-based format, while group papers often require images, which affects the relative file sizes. In calculating file counts, we must also differentiate between finals submissions and working drafts, each of which there may be many for a single assignment for a single student or team.

In the following formula, let Q be the number of recitation questions, C the number of critiques, and R the number of peer reviews. For team assignments, P represents the DPPR, D the DPR, and V the video. Each of these is an assignment with a submission through SubMIT. The coefficient of each variable is the expected size, in megabytes, of each submission.

$$\#Students\,(0.02Q + 0.1C + 0.06R) + \#Teams\,(0.3P + 0.6D + 100V) =$$
$$400\,(0.02*25 + 0.1 + 0.06) + 150\,(0.3 + 0.6 + 100) = 15399\,MB \cong 16\,GB$$

[1] https://www.zdnet.com/article/how-to-disk-partitioning-for-linux-and-windows-dual-booting/

We can see that video storage is the most significant portion above, accounting for about 96% of total student submission storage allocation. Fortunately, it is still only a fraction of the total available disk space. Lastly, we were able to build model SQLite databases on our own machines to estimate the overhead and cost of data storage for our three SubMIT databases. The assignments database should have about 40 entries. The submissions database should have about:

$$\#\,Students * \#\,Individual\ Assignments + \#\,Teams * \#\,Team\ Assignments =$$
$$400 * 36 + 150 * 4 = 15000\ entries$$

The video rankings database should have about 150 entries. Modeling these numbers with dummy data, the total size is on the order of several megabytes. In other words, these databases will not require significant storage resources.

In conclusion, expected storage on disk is about 25 GB for one semester's worth of material. This gives us a significant buffer in case of underestimation.

Computing Resource Allocation

There are five main processes the SubMIT server will need to handle: processing shared file syncs, processing incoming grades and comments, retrieval of files, grades, and comments, processing miscellaneous tasks like applying late penalties and late drops, and processing assignment submissions. Among these duties, the last--processing assignment submissions--differentiates itself as being critically high priority. Processing or retrieving files, grades, and comments is a next priority insofar as users expect these tasks to complete in a reasonable amount of time. Miscellaneous tasks, finally, need only happen within some reasonable amount of time.

To manage and prioritize requests from clients, SubMIT implements a request queue for processor allocation. The request queue prioritizes assignments submissions, then level-two priority tasks, and finally miscellaneous operations. By coordinating with the SubMIT web interface, requests and uploads are not initialized until the server has processors available. In addition, within ten minutes before a deadline, at least two processors are specially dedicated for receiving incoming submissions, ensuring that processors are not fully occupied at this time.

Data Transfer Protocol

Utility is one of the central design goals of SubMIT, and a speedy, responsive user experience is certainly one component of utility. For this reason, we designed the SubMIT data transfer protocol (DTP) to best manage the types of data streams being uploaded to and downloaded from the server. In designing the DTP, one of the first considerations was simply: what kind of files and data will we be transferring? Secondly, how reliable does the transport of these data need to be? Some files, like videos, will be on the order of hundreds of megabytes while others, like pulling a single assignment grade and comments, might be on the order of only kilobytes. Transmission Control Protocol (TCP) makes a suitable choice of underlying transport protocol, as loss of portions of assignment submissions or grades is not acceptable for the SubMIT system. Guaranteed in-order delivery of packets from sending to receiving application, even at the cost of some speed, is a necessary feature of a consistency-critical system like SubMIT.

After choosing TCP, another essential question was the trade-off over the size of packet payload, where decreased payload size increases metadata overhead, particularly for small data transfers, while increased payload size magnifies the effect of packet loss on transfer latency. In analyzing the asymptotic runtime for a file transfer, where *n* is the number of bytes to send and *L* is the loss rate of the network, we arrive at the following recurrence for transfer time:

$$T(n) = (1 - L)T(n) + O(n)$$

For TCP, we estimate an average loss rate of 5%; however, for any $L < 1$, this recurrence is linear in n. This indicated to us that perhaps the risk of packet payloads being too small is more significant than the risk from loss of large packets. With the total size of the IP header (~20 B) and TCP header (~20 B) coming to 40 bytes or 0.04 kilobytes, a packet payload size of 1 kilobyte would give a payload to header ratio of about 25:1. Considering, as well, that the smallest data transfers for SubMIT will happen on the scale of kilobytes, SubMIT segments data transfers into chunks of about one kilobyte for each packet payload.

In addition to considering packet size, it is important to consider what data is sent, and when. While TCP ensures packet delivery and ordering, the transport layer has no notion of abstract data structures like assignments or submissions. For uploads to the server, the SubMIT DTP specifies three stages of data transfer between the client through the SubMIT website and the SubMIT server: information about the request, a file transfer (if applicable), and a confirmation. The first stage can usually be encapsulated in a single packet. At a minimum, all information stage packets contain the authenticated Kerberos of the user, the request type (an integer, e.g. 0 for assignment creation, 1 grade retrieval, ...), and the time of sending. Depending on the type of request, there may be auxiliary information included in this packet, like information to include in a database entry if applicable to the request. In the file transfer stage, any documents or videos associated with a request will be uploaded. The final packet sent in this upload will set the FIN TCP flag to 1 to signify finality. When the SubMIT server receives this packet, we enter the final DTP stage: confirmation. In addition to sending an ACK, the SubMIT server sends a confirmation packet to the client, confirming that the transaction specified by their request is complete.

If the request appends or modifies a database entry, we treat the entire data transfer as an atomic unit. By default, information stage packets which induce a database appension or modification will set the “completed” bit for that entry to 0. In the case that a file upload fails part way through, there will exist a database entry recording the user, time of submission, and other details; however, the transaction will not be confirmed to have completed. If a transfer is taking much longer than expected (i.e. the completed bit is 0 after some extended amount of time), SubMIT will send an email notification, in addition to displaying so on the website dashboard, that the request/upload failed.

Evaluation and Use Cases

Late Drop

Given that any student within 6.033 has the option to drop the class as the semester progresses, the Administrative Staff must update the file system to reflect that the student is no longer able to participate in the submission/grading process. Our hierarchical file system allow the staff to easily remove an unregistered student from our server through a function provided by our system: *dropped_student(kerberos)* which removes the student's associated directory within *all_students,* all symbolic links to the directory across the file system, and all of the student's corresponding submission and grading database entries. Any updates pulled from Gradescope that contain information of the dropped student will not affect the system, since the associated database entries will be removed and Gradescope data will be unable to sync with the database. All read and write permissions are also revoked from the student by MIDS to prevent any unintended or malicious overwrites to individual and team assignments.

Bulk Account Creation

At the beginning of the term, accounts are set up for all students that are registered in the class through the function *make_student_dirs(class_list).* This populates the *all_students* directory with a folder for each student's associated kerberos, wherein each student folder contains directories for each individual assignment based on the entries within the assignment database. Students are also given read and write access to their own directory through MIDS.

For recitation section assignments, we assume that staff handles the logistics of scheduling students into recitation sections separately from SubMIT and provides a student : recitation map to another function, *make_recitation_dirs(recitation_map).* This populates the root folder with a directory for each recitation section, which contains symbolic links to the student directories in *all_students* that are assigned to that recitation section.

Since both functions are only creating directories for the submission function and creating permissions, setting up all of the student accounts at the beginning of the semester should take a minimal amount of time. Furthermore, we assume that the course lecturer will have communication with MIT's registrar to get this list soon, and begin the process before the term begins. This allows for the simple process of taking out a small number of students who do not take the class versus waiting for the class list to be figured out, and to jam server traffic at the start of the term with populating the directories.

Bandwidth & Speed

A central challenge with managing bandwidth and designing for a speedy user experience is managing bulk submissions at the deadline. We focus on this worst case because, in designing for it, SubMIT will be able to handle lower levels of throughput quite easily. Imagine it is 11:55 PM, and 400 students are submitting system critiques about 100 KB in size. The bottleneck throughput in the transport network is most likely edge device upload speed at 500 Mb/s. At this speed, a system critique would upload in:

$$100\ KB = 0.8Mb * \frac{1}{500}\frac{s}{Mb} = 0.0016\ s = 16\ ms$$

The SubMIT server processors have a clock speed of 2.1 GHz, so under the assumption that the processors are pipelined but occasionally stall, we assume we can read or write one bit per two cycles per processor, or 1.05 billion bits per second per processor. In 16 milliseconds, this gives:

$$1050000000\frac{bits}{s} * \frac{1}{1000}\frac{s}{ms} * 16\ ms = 16800000\ bits = 2.1\ MB$$

Thus, even with some variation in actual performance, dedicating one processor per upload should be sufficient. Total expected time spent uploading therefore is:

$$16\ ms\ *\ 400\ =\ 6400\ ms\ = 6.4\ s$$

By leveraging the SubMIT computing resource allocation scheme, even if everyone submits right be the deadline, the server should be able to handle the submissions.

In designing SubMIT, we recognize that brown-outs are an uncontrollable effect of the network at MIT. We assume that students will not be overly concerned if speed of interactions with the system decline for a short period of time. Additionally, we assume that staff can acknowledge the time of a brown-out. In the event of one close to submission time, we provide the course lecturer with the ability to update the Assignment Database with an extension of time. We do not address a solution to this problem more aggressively as we believe that it does not severely affect our design goals. We rely on course staff to handle edge cases in these situations as we aim to provide simplicity for clients, not efficiency.

Scalability

We design SubMIT in a way that supports applications to multiple classes across multiple MIT departments. SubMIT possesses an understandable architecture that aims to provide clients with simplicity of interaction from multiple aspects of submission and grading. The wide range of assignments in 6.033 constructs a sturdy framework for similar assignments encountered in other classes. The ability to include/exclude assignments from the database allows for the adjusting of the framework for which student files will be created. By designing SubMIT to meet the design goals of simplicity, security, and utility, we achieve scalability and applicability to other classes.

Threat Model

Security is one of our central design goals for SubMIT, as we want to ensure our users are using a reliable system that protects from malicious grade manipulation or submission edits/deletions. This is achieved by our handling of permissions through our MIDS implementation that follows the guard model. To ensure that only students and staff can access SubMIT, we modify our MIDS implementation to store salted hashes of user passwords, so adversaries can't breach our server through the construction of a rainbow table. This is under the assumption that students and staff act ethically and proper oversight of the system is enacted to prevent misuse.

Conclusion

SubMIT utilizes a dual file system hierarchy and structured databases with limited access via security at multiple levels, to provide students and staff with a simple system that handles all work and supports staff feedback for grading. By limiting access to files and utilizing our checkout system for student assignment work our system enables students to confirm confidence in the safety and privacy of their work. Furthermore, our SubMIT server hides many of the logistical complexities from students such as dividing them up into recitations and tutorial sections. With the structure and modularity of SubMIT server we deliver a *simple* and *secure* framework for students, while still providing staff the tools necessary to address needs for operation of the class, and overall *utility* for both users. Future iterations of SubMIT will uphold current design principles and develop more *efficient* and *robust* storage and sync services to improve the quality of users utility.

Individual Contributions

As a design team, we collaborated to iteratively construct our system's architecture, ensuring design goal motivations for design choices. Russell blueprinted and implemented student interactions with the system, Christian designed staff interactions with the system, and Jeremy modeled server processing and data transfer protocol. All team members worked on integration of modules and overall system evaluation.

Acknowledgments

We would like to thank Mohammad Alizadeh and Steven Okada for technical feedback on our design, Juergen Schoenstein for his guidance on our written report, and Katrina LaCurts along with the entire 6.033 staff for their teaching and instruction.