WatDFS: A Project for Understanding Distributed Systems in the Undergraduate Curriculum

Michael Abebe, Brad Glasbergen, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{mtabebe,bjglasbe,kdaudjee}@uwaterloo.ca

ABSTRACT

The ubiquity of distributed computing systems has led to an increased focus on distributed systems in the undergraduate curriculum. In this paper, we describe the design of, and our experiences with, a new distributed systems project where students implement the core components of a distributed file system called WatDFS. The WatDFS project enables students to meet learning objectives from across the distributed systems curriculum and interact with real systems, while providing a high-quality testing environment that yields actionable feedback to students on their submissions.

ACM Reference Format:
Michael Abebe, Brad Glasbergen, Khuzaima Daudjee. 2019. WatDFS: A Project for Understanding Distributed Systems, in the Undergraduate Curriculum. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287473

1 INTRODUCTION

The collection of vast amounts of data in domains such as science, commerce and health has outstripped the computing and storage capacity of a single, centralized machine, fueling the need to store and access these data on clusters of machines that together provide enhanced storage and compute capacities [30, 33]. Many everyday services that users interact with, such as Netflix, Gmail and Facebook are all backed by such clusters. These clusters are popularly called *distributed systems* and have become ubiquitous, providing the view of a single coherent system with appropriate transparencies for storing and managing user data. Due to their ubiquity, from the perspective of both individual users and the design and development of large-scale industrial applications, there is a need to focus on distributed systems in the undergraduate curriculum [4, 24].

At our university, distributed systems is a semestered course taught at the 4th-year level to approximately 120 undergraduate students, with an operating systems course being a pre-requisite. The course focuses on the fundamentals of distributed systems [34], covering the following topics: architectures of distributed systems, a brief overview of computer networks, communication paradigms including remote procedure calls (RPCs), distributed file systems, coordination/synchronization, replication, consistency, and fault tolerance. A primary focus of the course is understanding the trade-offs made as a consequence of design decisions in distributed systems [3, 8, 23]. We evaluate a student's understanding of these fundamental concepts and their trade-offs via four assignments throughout the semester and a final written exam. Two written assignments focus on assessing the students' understanding of the principles of distributed systems. The other two assignments together form a programming project that enables students to apply the fundamental course concepts through their *implementation and use*. These programming assignments, and our experiences with them, are the focus of this experience report.

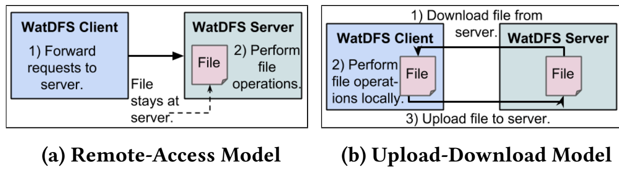
Previously, a distributed systems programming project in our course required students to implement the functionality of an RPC library using socket primitives [7, 16] to support client-server computing. To allow students to learn the basics of socket programming, students first implemented an `echo` server that would respond to client requests with the contents of each request. For the second programming assignment, they developed an RPC library that allowed applications to invoke function calls on a remote server. This second assignment required locating a server that could satisfy client requests via a directory server, serializing the request, and sending it to the remote server. The server deserializes the request, executes it, and then serializes and sends back the result.

Reflecting on the previous assignments that comprised the programming project, we observed that the assignments covered topics from only the first few course modules. Furthermore, the product of the assignments was an RPC library that necessitates integration into a software project from its inception. Consequently, students lacked the opportunity to use their library with real-world system software and applications that use RPCs. Finally, some students made small mistakes in their assignments which stymied communication between client and server, hindering the implementation and testing of the rest of the assignment's functionality. Based on these reflections, we decided to develop an entirely new set of programming assignments that encompass material from the whole course, allow students to interact with existing systems and applications seamlessly, and support a testing environment that gives students high-quality and timely feedback on their submissions.

Towards this goal, we created a programming project that comprised a pair of assignments in which students develop a distributed file system, **WatDFS**, which acts as a *real* file system in a Unix environment. Consequently, students can interact with their file system using standard tools and Unix commands such as `cat`, `echo` and `cp` to access and manage their files, or edit them with a text editor.

Students implement WatDFS using the Filesystem in Userspace (FUSE) library [2, 35, 36], which is supported by the Virtual File System (VFS) layer of most popular operating systems. A key advantage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5890-3/19/02...$15.00
https://doi.org/10.1145/3287324.3287473



(a) Remote-Access Model **(b) Upload-Download Model**

Figure 1: Two models of distributed file systems: the remote-access model, which sends all requests to the server, and the upload-download model, which operates on files locally.

of using FUSE is that it precludes the need to write kernel-space code that is often complex and difficult to comprehend [26, 31] for students who want to learn the core concepts of distributed systems. Over the course of the two assignments, students implement WatDFS using two standard distributed file system models: the remote-access model, and the upload-download model [34].

In this paper, we share our experiences of rolling out the WatDFS assignments in the hope that our explanations and reflections will be useful pedagogically for other educators who design similar assignments or build upon our work in their classes. Importantly, our analysis focuses on how the WatDFS project serves to bridge the gap between the theory and practice of distributed systems, which is otherwise difficult to achieve in the classroom [5, 9, 11, 16, 25, 38].

2 DESIGN OF THE WATDFS ASSIGNMENTS

In the WatDFS assignments, students implement the core components of a distributed file system. Completed, WatDFS supports creating, opening, reading, writing and closing files on local or remote machines. WatDFS follows the typical client-server architecture: an application interacts via standard I/O system calls with a WatDFS client, which in turn communicates with a WatDFS server.

Students implement WatDFS using two common models in distributed file systems: remote-access and upload-download (Figure 1). In the remote-access model, the client forwards all requests to the server, which then performs the file system operations on behalf of the client. The remote-access model performs poorly if the latency between the client and server is high because every request must be sent to the remote (and potentially distant) server. By contrast, in the upload-download model, the client downloads a copy of the file from the server, performs operations locally and — when the client is done with the file — uploads the file back to the server. Consequently, the upload-download model reduces latency when repeatedly operating on a remote file by performing operations locally and avoiding excess communication with the remote server. More generally, the remote-access and upload-download models represent two common design choices in client-server architectures: moving computation to the server (in this case, file operations), or moving data (transferring files), and therefore computation, closer to the client. As students implement these models, they are directly presented with the algorithmic tasks and ensuing performance trade-offs discussed in the course.

2.1 Architecture of WatDFS

As mentioned in Section 1, we designed the assignments such that students can use and test their code using real systems and applications. In the course, students learn that popular distributed file systems are implemented such that applications can interact with them

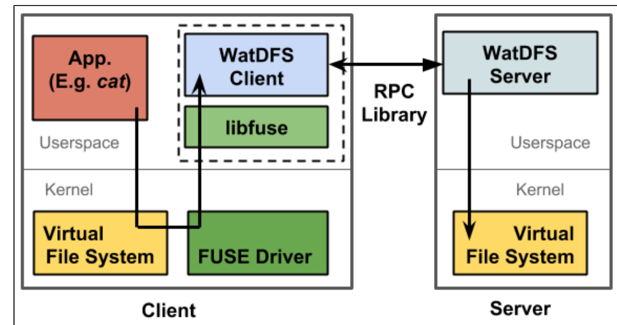


Figure 2: The architecture of WatDFS: requests from an application arrive at the WatDFS client through `libfuse`. If necessary, the WatDFS client communicates with the WatDFS server on a remote machine using an RPC library which performs file system operations on the server. Students implement the WatDFS client and server.

transparently. That is, applications operate without knowledge that the file they are accessing may reside on a remote machine [18]. In implementing the assignments, students directly observe the value of access transparency in file systems because they can use their assignment solution as a real file system without modifying existing applications. Although access transparency simplifies the integration between a distributed file system and applications, the file system may return errors, for example network errors, that applications do not expect. Thus, the WatDFS assignments also expose students to transparency challenges.

Traditionally, distributed file systems such as Sun's Network File System (NFS) achieve access transparency by integrating directly into the kernel. However, we preferred that students did not have to modify kernel code because of the challenges associated with the lack of memory protection, standard libraries, debugging tools, and complex dependencies [26, 31]. Instead, WatDFS uses FUSE to allow students to create their own file systems entirely in user-space in C or C++ (Figure 2). Applications issue system calls to manipulate files, which are passed through the kernel into the VFS and the FUSE driver. The FUSE driver issues the corresponding `libfuse` calls to a user-space client library, which the students implement. To manage files, the client library issues remote procedure calls (RPCs) to the WatDFS server that students implement. The WatDFS server handles client requests and supports ten `libfuse` calls for both assignments [2]. The semantics for the `libfuse` calls varies per assignment; we describe their behavior in the following sections.

We provide students with a custom RPC library, which supports registering functions at the server that the client can then call. The RPC library is provided to students to reduce implementation time and enables them to focus on distributed file-system concepts.

2.2 Remote-Access Model

The WatDFS assignments require students to support distributed file-system functionality using `libfuse` and an RPC library, both of which are new technologies for students. With that in mind, we designed the remote-access assignment to familiarize students with these technologies and the basics of WatDFS before beginning the more challenging upload-download assignment.

In the remote-access model, the WatDFS client handles `libfuse` calls by issuing file-system operations to the WatDFS server, which executes them on the client's behalf and sends back the results. We suggest that students first implement the `getattr` `libfuse` call, which is used to retrieve file metadata. Afterwards, students support the `mknod` call, which creates a file if it does not exist, and the `open` call, which opens a file and stores a file handle for use in subsequent `libfuse` calls. The `read` and `write` `libfuse` calls are considered next, which read data from a file into a buffer and write data from a buffer into a file, respectively. Then, students complete `release`, which is called asynchronously after a file is closed and signals that the file system can close the file and free any associated metadata. Finally, students implement the remaining calls: `utime`, which changes the file modification time; `fsync`, which flushes the file to disk; and `truncate`, which adjusts the file's size.

Students are provided with detailed descriptions of the `libfuse` calls in the assignment specification, and must decide which file-system operations the WatDFS server should perform when it receives a request from a client. To assist students with this task, we provide an instrumented `libfuse` library that logs the name and arguments of received calls to a debugging file. With the logged information, students can write or use existing applications to interact with their WatDFS client and observe how, and when, different `libfuse` calls are issued. Students are encouraged to use the `strace` tool [1] on existing applications (e.g., `cat`) to observe the system calls that they make and how they are translated to `libfuse` calls.

We simplified design and debugging for this first assignment by requiring students to follow a specific RPC protocol, which allows automated tests (Section 2.4) to quickly point out communication errors before students work on the higher-level functionality.

2.3 Upload-Download Model

In the second assignment, students implement WatDFS using the upload-download model. In the upload-download model, the WatDFS client downloads and locally caches a copy of the file from the WatDFS server, performs file operations on the cached copy and uploads the file copy back to the server when the client is done using it. Students build on their remote-access model solution, adding support for file transfer, cache consistency, and mutual exclusion.

WatDFS uses a timeout-based caching scheme similar to that of NFS, which allows stale reads, as covered in class. In this timeout-based caching scheme, a file is periodically re-downloaded from the server, or re-uploaded to the server, when a *freshness condition* F no longer holds. WatDFS defines the freshness condition for a file as follows.

Let T_{client} be the time the file was last modified by the client, and T_{server} the time the file was last modified at the server. Let $T_{validate}$ be the time the client last validated the freshness of the file, where $T_{validate}$ is the time the client downloaded the file if it has not yet been validated. Then the freshness condition for time T is $F = [(T - T_{validate}) < T_{fresh}] \vee [T_{client} == T_{server}]$, where T_{fresh} is a configurable interval that defines the amount of time that can elapse between file validation checks. When F is false, the client must synchronize its copy of the file with the server.

In words, the freshness condition is true, if the file has been recently validated, or if the file has not been modified locally at the client or the server since the time the file was last downloaded. At the end of freshness check, $T_{validate}$ is updated to the current time. Note that the first clause of the freshness condition can be evaluated without contacting the server and the second clause requires retrieving only file metadata from the server. Students are expected to include these optimizations in their solution.

Students are required to perform all file operations locally at the client, which means that a copy of the file must be downloaded first. Before every read operation, the freshness condition must be checked; if it fails, then the file is downloaded from the server. After write operations, the freshness condition is checked; if it fails, then the file is uploaded to the server. In WatDFS, three `libfuse` calls have special semantics: `open` should always fetch the file from the server, `release` should upload the file to the server if it was open for writes, and `fsync` forces an upload of the file to the server.

Students must prevent concurrent file modification by ensuring that a file is open in write mode by only one client at a time. The WatDFS server enforces this requirement by returning an error (`-EACCES`) if it receives a request to open a file in write-mode that has already been opened in write mode by another client. To prevent partial file updates, students implement atomic uploads and downloads of files, while allowing multiple simultaneous downloads. To assist students with these mutual-exclusion requirements, we provide them with an implementation of a reader-writer lock.

2.4 Assessment of WatDFS Assignments

Student submissions are primarily assessed through automated test cases. Students may submit their assignments any number of times before the deadline to a Marmoset server [29], which has two classes of test cases. A student can immediately view the results of all *public tests* for their submission, but must pass all of the public tests before they can run the *release tests*. For a given submission, a student can see at most two failed release test cases and can release test their submission at most five times in a 24-hour window. The limited availability of release tests encourages students to re-examine their code when they fail a test, develop their own system tests and to start their assignments early [29].

In the remote-access model assignment, the public tests issue simple file operations and verify that the RPC requests and responses are correct. These basic tests account for 30% of the assignment's mark. This assignment's release tests are end-to-end system tests, verifying that applications using I/O system calls perform correctly. The most difficult tests involve reading and writing large files, which necessitates invoking multiple RPCs to ensure that entire buffers of data are read or written. The release tests account for 65% of the assignment's mark, and 5% is awarded for a short README.

All tests in the upload-download model assignment are end-to-end system tests. The public tests are similar to the remote-access model's release tests, but also ensure that files are correctly uploaded and downloaded from the server, and account for 40% of the assignment's mark. The release tests are worth 35% of the assignment's mark and test the cache consistency and mutual exclusion requirements of the assignment. For example, one test ensures that when the freshness interval expires, a client re-downloads the file from the server and sees any updates. Finally, we allocate 25% of the assignment's marks to a system manual, where students describe the architecture and design decisions of their WatDFS solution.

3 OUTCOMES AND EXPERIENCES

The WatDFS assignments are designed with a number of learning goals and objectives in mind. In this section, we discuss these goals, share our experiences from the deployment of the assignments in our course, and describe extensions to WatDFS.

3.1 Learning Goals and Objectives

The WatDFS assignments give students the opportunity to *use* and *apply* concepts from distributed systems by implementing the core components of a distributed file system. We highlight the relevant curricular content that students are exposed to when doing the WatDFS assignments, based on learning objectives in the distributed systems course and from the relevant curricula [4, 24].

3.1.1 Relevance to Course Material. The course introduces distributed system *models* and the *trade-offs* associated with them. In WatDFS, students implement a client-server model: the WatDFS client initiates a request on behalf of the application, and the WatDFS server satisfies the request. The remote-access model features a stateless client, while the upload-download model uses a stateful client. When implementing the upload-download model, students must manage *distributed state* and *replicas* using the timeout-based cache *consistency* scheme. A complete solution also uses *mutual-exclusion* primitives to ensure that there is at most a single-writer to a file, and to provide *atomic* file transfers during the file upload or download. Finally, for the WatDFS client to communicate with the WatDFS server, students must make *RPCs*. In summary, by implementing the two common models of distributed file systems, students are directly exposed to the trade-offs between the two models, which are representative of choices faced by distributed systems developers.

Given the nature of the WatDFS assignments, students also focus on *systems-related* learning objectives such as using file systems. While completing the WatDFS client, students must implement `libfuse` calls, which have similar behavior and interfaces to VFS calls. Students were also exposed to *access transparency* since the WatDFS client enables applications to access local and remote resources using the same file system interface.

3.1.2 Use of Real Systems and Applications. FUSE enables students to develop a distributed file system that supports access transparency. This design choice has two important consequences: (i) it exposes students to key systems and software development practices, and (ii) it makes it easier for students to test and reason about correctness (Section 3.1.3).

As WatDFS is a programming assignment, it requires students to apply many fundamental concepts in *system and software development*. We provide students with starter code and libraries to complete the WatDFS assignments. To successfully use this code, students must read and understand the documentation and examples, building on their experience from doing so in their prerequisite operating systems course. As WatDFS executes entirely in user-space, students had to perform system calls. Doing so requires understanding system call functionality, arguments, and return values. The ability to read and understand documentation and follow examples is an essential skill when working with systems/technologies [14]. In fact, it is a common requirement for students who are asked to work on system design and development on a work (co-op) term or when they start full-time employment as software developers. [1]

3.1.3 High Quality Testing Environment. WatDFS acts as a real file system, allowing students to test their code easily: any existing application that performs file creation, reads, or writes can be used to test functionality. Students are required to discuss their testing strategy in their documentation, which encourages students to design and write their own tests. Many students reported using existing command line tools (`echo`, `cat`, `less`, `tail`) to manipulate files. Students also wrote their own test cases using programming languages such as `C++` and `python` to edit files in WatDFS using language-specific features.

To provide students with feedback, we used Marmoset [29] as an automated test-server. As described in Section 2.4, we used a combination of public and release tests. However, for each failed test we also provided guidance on how students could alter their system. For example, in the case of the RPC protocol tests for the remote-access model, a test asserted that the WatDFS client issued the correct RPC and handled the return correctly. If the test fails, we configured Marmoset to print any called `libfuse` functions and their expected argument types to help students reproduce the test locally. For the end-to-end system tests, we provided a brief description of the failed test which a student can use to construct a similar test case as well as a general description of the cause of failure. For example, a test description in the upload-download model is "Testing error handling when a file is opened multiple times"; on a test failure a student would receive: "Expected to get `EACCES` on second file open, but got no errors". Such guided statements helped students correct their own code, and as we discuss next, resulted in a positive experience with the assignment.

3.2 Experiences with WatDFS

We now share our experiences with the WatDFS assignments, highlighting their effectiveness and providing insight into our plans to iteratively improve the teaching and learning environment for students. We base our discussion on analyses of 115 student grades, questions posed by students on the class discussion forum, and conversations with students both during office hours and in class.

Figure 3a displays a cumulative distribution function (CDF) of student grades for the remote-access model assignment. The x-axis of the graph corresponds to the percentage of students that achieved at least the grade on the y-axis. We show three plots, one for each of the three sub-components of the assignment. The figure shows that almost all students (95%) achieved full marks for the RPC protocol portion of the assignment, indicating that students understood how to use the RPC library and `libfuse`. Students who lost marks on the assignment tended to do so on a test of handling reads and writes from/to large files, which was worth 10% of the (total) assignment mark. Submissions that failed this test did not loop within `libfuse` `read` and `write` calls until the requested amount of data was read/written, as noted in the assignment specification.

To better understand how students worked through the assignment, we examined 221 student questions on the class discussion

[1] Our university has a large co-operative education program, and many students work at software/systems companies such as Google, Facebook, Amazon or start-ups.

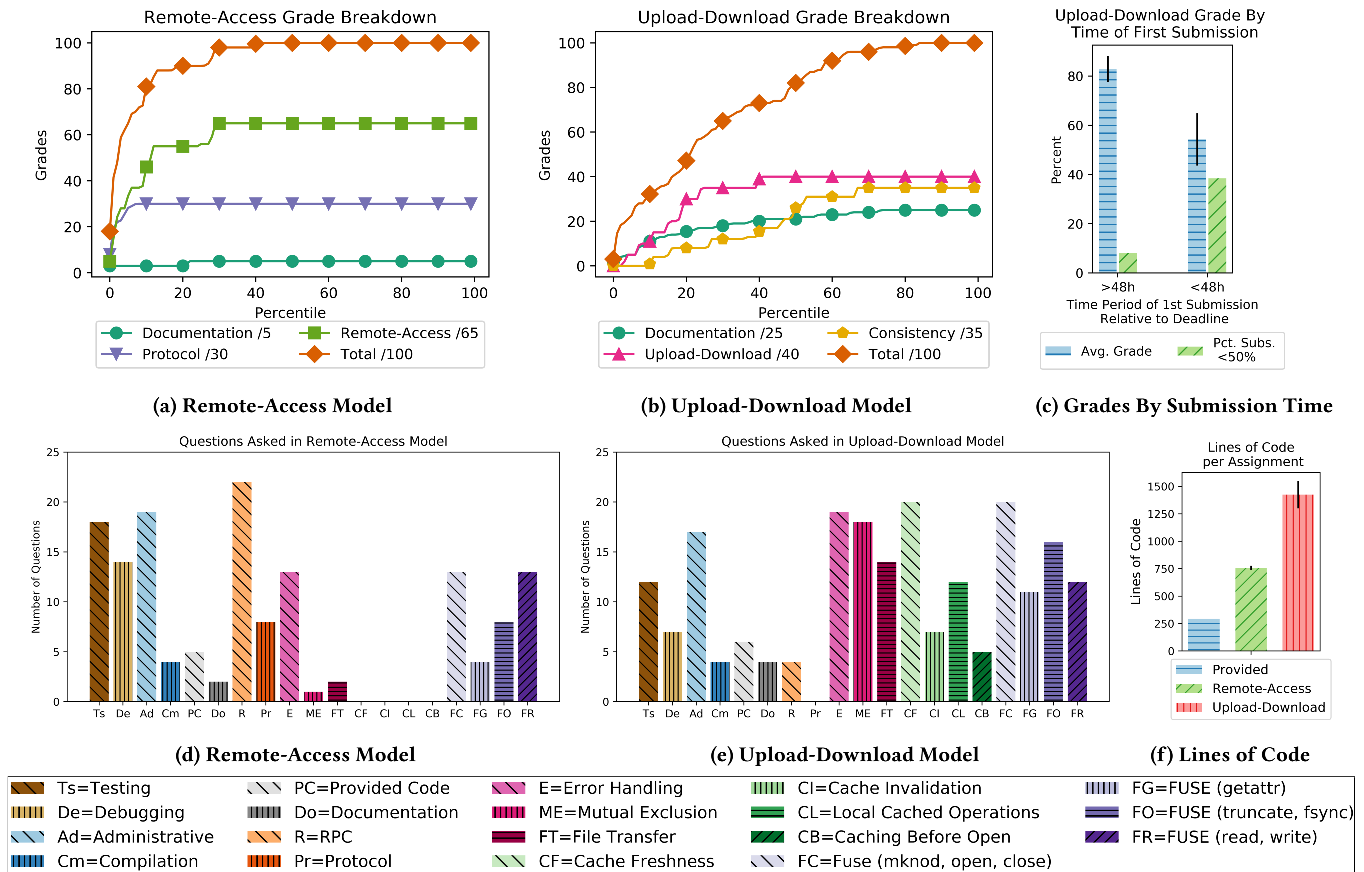


Figure 3: Metrics for student assesment in the WatDFS assignments. Student grades, displayed as a cumulative distribution function (CDF), for WatDFS assignments, and average lines of code. A break down of the number of questions asked in each assignment that relate to a particular topic. Sub-topics share a color, and a short description of each topic is offered as a legend.

forum. We selected 19 different categories and subcategories relating to the assignment specification/requirements and labeled each question according to which of the categories and subcategories it belonged. In Figure 3d, we display the number of questions that were asked in each category for the remote-access model assignment. Subcategories share a common color, and the first letter of their abbreviation, with their parent category. For example, questions about FUSE (purple color — FC, FG, FO, FR) were further categorized based on the specific function in question.

As noted in Section 2.2, the remote-access model assignment is designed to familiarize students with the RPC library, `libfuse`, and programming/debugging in WatDFS. Consequently, student questions for this assignment (Figure 3d) were primarily focused on testing and debugging (e.g. "Why am I failing this test?", "How do I trace the calls '`cat`' makes?"), the usage of the provided code (e.g. "Why am I getting this RPC error?"), or `libfuse` calls, specifically those related to file creation, reading and writing. Based on the types of questions that students asked and the grades that students attained, we can infer that the remote-access model was successful in helping students familiarize themselves with WatDFS.

In contrast to the remote-access model, the upload-download assignment generated questions that were mostly related to error handling, cache consistency, mutual exclusion and how these topics applied to operations supported by `libfuse`. Error handling was more important in the upload-download assignment than in the remote-access model because the remote-access model simply reported errors back to the application. In the upload-download model, students had to decide what to do when an error occurred in the middle of an operation, such as during file transfer.

Given the objectives of the upload-download assignment, many of the questions related to cache consistency: clarifying the freshness condition or when a file should be considered invalid. These questions focused on the semantics of caching in the context of `libfuse` operations that are outside the boundaries of file open and close operations. The crux of these questions is the gap between the theory and practice of the upload-download model. Specifically, standard textbooks [34] state that the upload-download model performs all operations locally, and that the file is first downloaded to the client on the file open call. However, downloading a copy of the file only on `libfuse` open while performing all file operations locally is not possible because some `libfuse` calls (e.g. `getattr`, `truncate`, `mknod`) can occur before open. Consequently, some students were unsure of the semantics because not all operations can be applied locally if the file is downloaded only on open. In future

versions of the WatDFS assignments, we plan to be clearer that all operations are to be performed locally, even if that necessitates downloading the file first to support the upload-download model semantics. We say this while keeping in mind that practice can deviate from theory where system design and development is concerned.

While reviewing student grades for the upload-download model (Figure 3b), we observed more variation compared to the remote-access model. A majority of students (60%) passed all of the basic upload-download model tests, with 33% of students passing all of the cache consistency tests. Overall, more than 80% of students passed the majority of the programming tests. Inspection of submissions that passed all of the upload-download tests, but did not pass all of the cache consistency tests (27% of students) revealed that these submissions either did not check the freshness condition and/or enforce mutual exclusion, or had errors in implementing these requirements. We gave students three weeks to work on the upload-download model and through weekly reminders emphasized the importance of beginning the assignment early. Indeed, we observed a significant difference in the average grade between students who had at least one submission to the Marmoset test servers 48 hours before the assignment deadline and those who did not (Figure 3c). 70% of students who received less than half of the programming marks had no submissions until within 48 hours of the assignment deadline, indicating that they may have run out of time. Moreover, 45% (8%) of students who had their first submission within (prior to) 48 hours of the deadline did not pass the majority of tests.[2] Figure 3f displays the average lines of code in student submissions for the remote-access (758 lines) and upload-download (1425 lines) assignments, including provided code (294 lines).

Finally, as noted in Section 2.4, 25% of the marks in the upload-download model were allocated for a system manual describing design choices. We encouraged students who did not complete the full functionality of the assignment to describe how they would have implemented the functionality in WatDFS. Importantly, this requirement allows students to express their understanding of the concepts even if they had a partial implementation.

Overall, our experience has been that the WatDFS assignments exposed students to the core aspects of distributed systems and key systems technologies. Although anecdotal, a student summarized their experience as: "*This assignment was probably my favorite part of the course and the most fun I've had programming for any assignment [...] FUSE was a brilliant way to integrate the course concepts into something very practical and I was really happy that I could actually test my work by trying to edit the remote files in Vim.*"

3.3 Extensions to WatDFS

There are several opportunities for extending the WatDFS assignments. Both the remote-access model and the upload-download model could be extended to include operations on file system directories, which students were not required to support. To do so in the upload-download model would require managing a cache of different object types with different rules for files and directories. The WatDFS assignment currently operates with all clients interacting with a single server. Students could be tasked with managing multiple WatDFS servers or using a quorum protocol to keep server state consistent while supporting file replicas.

[2] 65% of students had at least one submission more than 48 hours prior to the deadline.

With the exception of the `libfuse release` call, WatDFS operations are synchronous. Consequently, in the upload-download model, the entire file must be downloaded before any operations can take place. Students could implement asynchronous features in WatDFS to support downloading a file in the background, or predictively prefetching a file. Alternatively, WatDFS could support Coda file system style callbacks that indicate a file has been modified, and allow the application to force a new download of the file [27].

Finally, although WatDFS follows a design similar to NFS, there exist a wide variety of distributed file systems including those that provide varying consistency models [32], fault tolerance schemes [17, 37], metadata management [13], or interfaces [22]. The WatDFS assignments could be modified to incorporate these ideas.

4 RELATED WORK

The WatDFS assignments are most closely related to the user-level distributed file systems projects described by McDonald [20]. However, any application can use WatDFS as a file system because FUSE maintains standard file system interfaces. Thus, students use standard tools and language features to interact with WatDFS, as opposed to a limited number of rewritten applications [20]. Furthermore, WatDFS focuses on the consistency of distributed file systems and allows students to examine the differences between the remote-access model and the upload-download model. McDonald's assignment is in the context of a computer-networks class, so it focuses on the implementation of specific networking protocols [10]. Although there has been discussion on using `libfuse` for educational systems [12, 19, 21], WatDFS focuses on the different models of distributed file systems and core distributed systems concepts.

A typical operating systems course assignment is to develop file system functionality inside an educational operating system. Such assignments may implement an alternative file system design, ensure crash-consistency, or extend file system functionality with key-value stores [6, 15]. WatDFS differs by focusing on file access and management in distributed systems and the challenges that arise in a distributed environment. Finally other user-space systems assignments [28] for tasks traditionally done in kernel-space (e.g., memory management), share our experience that it is easier for students to debug and develop outside of the kernel while learning the core concepts and challenges associated with the task.

5 CONCLUSION

In this paper, we described our experiences with a new pair of programming assignments for a distributed systems class where students develop a distributed file system called WatDFS. The WatDFS assignments provide students with the opportunity to compare two popular models of distributed file systems, and importantly, use and apply concepts from the entire course. Our early analysis found that the first assignment successfully introduced students to the new technologies used in the assignment, and over 80% of students passed the majority of the test cases in the second, more challenging, assignment. We plan to use our experiences to continue to improve the teaching and learning environment for students. We hope that by sharing our experiences, other educators will be motivated to pursue similar projects and benefit from our insights.

ACKNOWLEDGMENTS

We thank Byron Weber Becker and Heather Root for their helpful comments on this paper, as well as Mikhail Kazhamiaka, Siddhartha Sahu and Zhiwei Shang who supported the deployment of the assignment as teaching assistants. The University of Waterloo provided funding for this project.

REFERENCES

[1] 2017. strace. https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/.

[2] 2018. libfuse. https://github.com/libfuse/libfuse.

[3] Daniel J Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 2 (2012), 37–42.

[4] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013.* Technical Report. ACM Press and IEEE Computer Society Press. https://doi.org/10.1145/2534860

[5] Jeannie R Albrecht. 2009. Bringing big systems to small schools: Distributed systems for undergraduates. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 101–105.

[6] Jeremy Andrus and Jason Nieh. 2012. Teaching operating systems using android. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education.* ACM, 613–618.

[7] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

[8] Navin Budhiraja and Keith Marzullo. 1992. *Tradeoffs in Implementing Primary-Backup Protocols.* Technical Report. CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE.

[9] Justin Cappos, Ivan Beschastnikh, Arvind Krishnamurthy, and Tom Anderson. 2009. Seattle: a platform for educational cloud computing. *Acm sigcse bulletin* 41, 1 (2009), 111–115.

[10] Martin Casado and Nick McKeown. 2005. The virtual network system. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 76–80.

[11] Henrik Bærbak Christensen. 2016. Teaching DevOps and cloud computing using a cognitive apprenticeship and story-telling approach. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, 174–179.

[12] Jeffery Forrester. 2015. Platforms for Teaching Distributed Computing Concepts to Undergraduate Students. (2015).

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. *The Google file system.* Vol. 37. ACM.

[14] Sarah Heckman, Kathryn T Stolee, and Christopher Parnin. 2018. 10+ years of teaching software engineering with itrust: the good, the bad, and the ugly. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training.* ACM, 1–4.

[15] David A Holland, Ada T Lim, and Margo I Seltzer. 2002. A new instructional operating system. In *ACM SIGCSE Bulletin*, Vol. 34. ACM, 111–115.

[16] Mark A Holliday, J Traynham Houston, and E Matthew Jones. 2008. From sockets and RMI to web services. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 236–240.

[17] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. 2012. Erasure Coding in Windows Azure Storage.. In *Usenix annual technical conference.* Boston, MA, 15–26.

[18] Steve R Kleiman et al. 1986. Vnodes: An Architecture for Multiple File System Types in Sun UNIX.. In *USENIX Summer*, Vol. 86. 238–247.

[19] Geoff Kuenning. 2011. CS 135 FUSE Homework Assignments, Fall 2011. https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/.

[20] Chris McDonald. 1996. User-level distributed file systems projects. In *ACM SIGCSE Bulletin*, Vol. 28. ACM, 333–337.

[21] Halli Elaine Meth. 2014. Decafs: A modular distributed file system to facilitate distributed systems education. (2014).

[22] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: FacebookâĂŹs warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation.* USENIX Association, 383–398.

[23] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. 2005. Design and implementation tradeoffs for wide-area resource discovery. In *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on.* IEEE, 113–124.

[24] Sushil K Prasad, Almadena Yu Chtchelkanova, et al. 2011. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates.

[25] Ariel S Rabkin, Charles Reiss, Randy Katz, and David Patterson. 2012. Experiences teaching MapReduce in the cloud. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education.* ACM, 601–606.

[26] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing.* ACM, 206–213.

[27] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers* 39, 4 (1990), 447–459.

[28] Sam Silvestro, Timothy T Yuen, Corey Crosser, Dakai Zhu, Turgay Korkmaz, and Tongping Liu. 2018. A User Space-based Project for Practicing Core Memory Management Concepts. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* ACM, 98–103.

[29] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin* 38, 3 (2006), 13–17.

[30] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel big data science and technology center vision and execution plan. *ACM SIGMOD Record* 42, 1 (2013), 44–49.

[31] Red Hat Storage. 2011. Linus Torvalds doesn't understand user-space filesystems. https://redhatstorage.redhat.com/2011/06/28/linus-torvalds-doesnt-understand-user-space-storage/.

[32] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review*, Vol. 29. ACM, 172–182.

[33] Maarten van Steen and Andrew S Tanenbaum. 2016. A brief introduction to distributed systems. *Computing* 98, 10 (2016), 967–1009.

[34] Maarten Van Steen and Andrew S Tanenbaum. 2017. *Distributed Systems.* Pearson Education.

[35] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of FAST'17: 15th USENIX Conference on File and Storage Technologies*. 59.

[36] Ivan Voras and Mario Zagar. 2006. Network distributed file system in user space. In *Information Technology Interfaces, 2006. 28th International Conference on.* IEEE, 669–674.

[37] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation.* USENIX Association, 307–320.

[38] Joel Wein, Kirill Kourtchikov, Yan Cheng, Ron Gutierez, Roman Khmelichek, Matthew Topol, and Chris Sherman. 2009. Virtualized games for teaching about distributed systems. *ACM SIGCSE Bulletin* 41, 1 (2009), 246–250.