A Distributed Grading System: How I Learned to Stop Worrying and Love the Grades My Peers Give Me

Final Project, CS262, Spring 2012

Team: Rob Bowden, Jim Danz, Tony Ho, Stefan Muller, Leora Pearson, Willie Yao, Kenny Yu

Introduction (What we did)

Problem Statement & Motivation

In large courses, grading requires a great deal of time and effort on the part of the course staff. The process of grading is necessary to evaluate the work of the students, but is otherwise a fairly unproductive exercise and consumes resources that could be more effectively used to enhance instruction. This problem is exacerbated in large-scale open courses such as those in the EdX program recently announced by Harvard and MIT. In these courses, which are designed to scale to arbitrary numbers of students, it is impossible for all of the grading to be done by course staff members.

These problems are solved by peer grading, in which students evaluate the work of other students. Not only does peer grading free up the course staff, but reading the work of other students will also improve students' understanding of the course material. Peer grading, however, involves its own set of logistical problems. For example, submissions and grades must be suitably anonymized to avoid conflicts of interest and privacy violations. In addition, for each assignment, submissions must be assigned to graders fairly, so that no student grades his or her own assignment. Finally, the quality of grades must be monitored so that students giving unfairly high or low grades can be identified. These students' scores can be weighted less in calculating the final grade of the assignment.

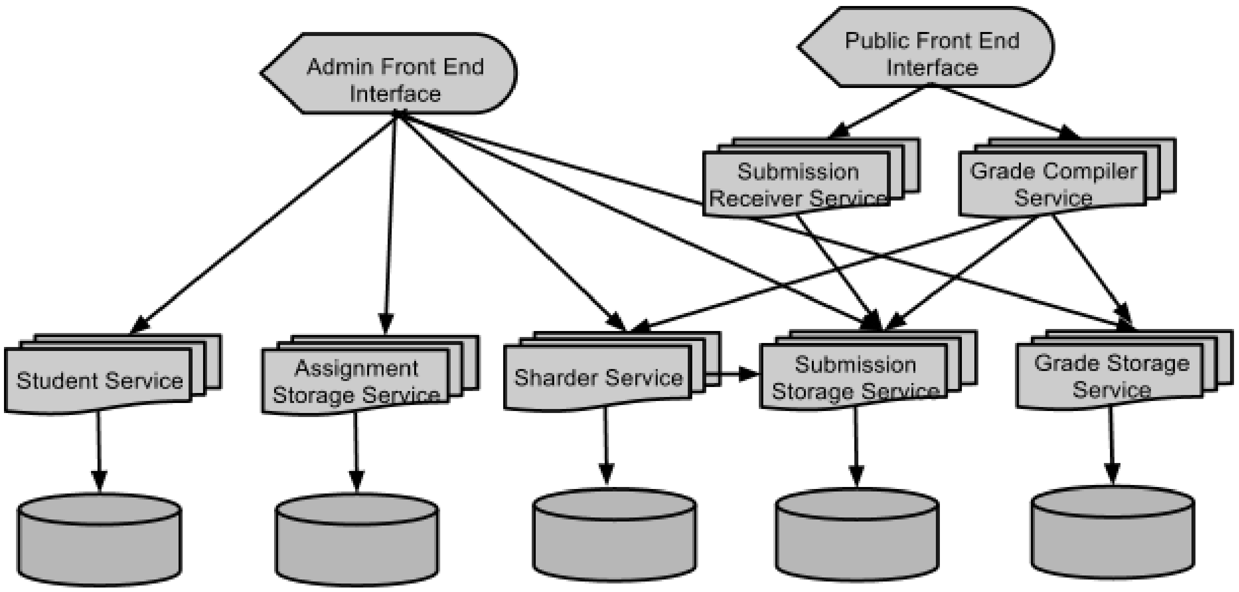
We have designed and built a distributed system to handle many of these logistics for a course staff. While this problem does not, at first glance, seem to lend itself to a distributed system, this system can be made more reliable by building it in a distributed fashion. For example, separating the different components allows students to submit and view grades even if the component that handles assignment submission is down. In addition, the distributed nature of the system allows for replication of services, which will also increase availability.

Grading Workflow

An admin (instructor) begins an assignment cycle by creating a new assignment and uploading a description using a web interface. Students may then complete the assignment and upload their submission online using an anonymized student ID. Once all of the student submissions are received, an admin may view the submission using the admin interface and can generate a "shard," which maps students to assignments which they are to grade. Assignments may be graded by multiple students. Each enrolled student is then sent an e-mail with the submissions they are to grade. When they are finished grading, students log in to the student web interface and submit their scores and comments. Students may then view all of the grades for that assignment, listed by anonymized student ID. Being allowed to view all of the submissions and grades will help students learn from their peers and also help ensure consistency in grading.

System Design (How we did it)

Service-Oriented Architecture & Persistent Storage


Admin Front End Interface
Public Front End Interface
Submission Receiver Service
Grade Compiler Service
Student Service
Assignment Storage Service
Sharder Service
Submission Storage Service
Grade Storage Service

We designed our system using a service-oriented architecture and made two assumptions about our system: our system is synchronous (to allow for use of time-outs and thus, eventual progress), and can only tolerate crash failures. Our system consists of three tiers: a user-facing front end, a number of stateless services that perform processing, and persistent storage provided by MongoDB.

The main front end is web-based and has two layers: the servlets and the servlet pages. The servlets are pure Java, and they sit at the edge between the server and purely client-side components of the system. The servlets communicate with the servlet pages via HTTP and with the rest of our system via RMI. For simplicity, each servlet performs one task, such as getting the grades for a particular submission. The user-facing web pages contain all the client-side programming (HTML, Javascript, etc.) and require little or no Java.

As of now, there are three groups of servlets and a corresponding servlet page for each. They provide interfaces for the three types of users that would interact with the system: students, the public and administrators (i.e., teachers). From the public page, users belonging to either of the other two groups can get to a login page by which they can identify themselves; however, at the moment, no real authentication is done. Ideally, we would tap into the Harvard PIN System, and automatically identify and redirect users who successfully login via the Harvard PIN login to the appropriate user page based on class and staff membership.

All of the services are implemented in Java and communicate with each other using Java RMI. These services and their descriptions are as follows:

- **Student Service**: Reports information (e-mail address, name, etc.) for students enrolled in the class. This may communicate with a school-wide database or another database kept up-to-date with course enrollment.
- **Assignment Storage Service**: Stores information (a unique ID and description) for each assignment in the class.
- **Submission Receiver Service**: Accepts submissions from the public front end interface and sends them to the Submission Storage service to be stored. This could be used to do automated tasks on submissions, such as running tests on student submissions.
- **Submission Storage Service:** Stores submissions for all assignments in persistent storage.
- **Sharder Service:** Upon request, shards assignments and assigns graders. The shard is written to persistent storage.
- **Grade Compiler Service:** Accepts grades (scores and comments) from the student interface and sends them to be written to persistent storage. This could also do more complex processing like combining and weighting.
- **Grade Storage Service:** Writes assigned grades to persistent storage.

We designed our system to maximize modularity, which will ease maintenance and increase reliability. For example, storage services that use different storage mechanisms or protocols can be easily switched in and out without taking the other services offline. In addition, if some components of the system fail entirely, other features of the system that do not depend on these components will remain available.

A major feature of these six services is that they maintain no state (though they may be associated with the MongoDB instances, which maintain persistent state). This allows us to achieve crash-failure fault tolerance easily through replication. Multiple replicas of each service are online at any given time, and a component that needs to connect to a service may use any available replica. If a replica fails, another replica may be used without any loss or inconsistency of data.

All of the persistent state is stored in five separate databases. For this purpose, we use five MongoDB clusters, each cluster individually managing its own replication. Each service is responsible for managing its own associated MongoDB cluster. We chose MongoDB because MongoDB is a distributed and reliable key-value store that allows us to easily store the documents from submissions. MongoDB uses replica sets, a form of primary-backup, to achieve replication and fault tolerance. By making each datastore orthogonal with the other datastores, a complete failure in one service will not affect the availability of the other services. For example, even if the submission storage service completely fails, students will still be able to submit grades.

Fault Tolerance

By using stateless services, we can easily create as many replicas as we need of the same service without having to coordinate between replicas. Thus, with *n* replicas of each service, each service can individually tolerate up to *n - 1* crash failures (only one replica is required to be up at any time). When a service A wishes to communicate with some other service B, service A can select any currently running instance of B (through our discovery process described below) and communicate with it.

For persistent storage, we use MongoDB's replica set replication model to achieve fault tolerance within each individual MongoDB cluster. For each individual cluster, we can launch new MongoDB instances to increase the number of replicas and thus, increase availability. By replicating all five of the individual datastores separately, we decrease the probability of multiple components of the system being unavailable at the same time. We favored this model over a model in which there is a single datastore shared by five storage services; a failure in this datastore would lead to unavailability of the entire system.

Discovery

Services find each other through a configuration file (located in the config/services.config) directory. The configuration files have the following form, where a service is followed by a DNS host and port containing a replica of that service:

```
#Service_Name_1
host1:port1
host2:port2
#Service_Name_2
host1:port1
host2:port2
host3:port3
…
```

Whenever a service wishes to communicate with another service it goes through the following steps:

1. Parse the configuration file
2. Looks for hosts containing replicas of the other service
3. Selects at random a host and port combination (default is 1099, the RMI default)
4. Attempts to communicate with the RMI registry on that host/port
5. Once the service has connected to the remote registry, the service does a name binding

lookup, using the desired service as the name, to retrieve a reference to a remote object.

6. If at any point we receive a failure (in the form of a RemoteException or NotBoundException), the service selects another host and port combination and repeats the process.

As an optimization, we only parse the file once (step 1) and load into memory a mapping between service names and hostnames containing replicas of that service.

In addition to the service locations, we also maintain a list of all currently running MongoDB instances for each cluster associated with a given service. Whenever we launch a persistent storage service, the service connects to all the instances of the cluster (MongoDB takes care of connecting with the primary node).

One advantage to this discovery mechanism is that adding new services to our system is relatively easy--all one would need to do is add a new entry to the configuration file. One downside to this discovery mechanism is that the topology of the entire system should be known before we start our services because rolling out changes to all replicas will be difficult. We must know the topology of the system because the configuration file must contain the hostnames for all known replicas of every service. However, if we do decide to change the contents of the configuration file once the services have started running, we can incrementally stop each replica and restart a new replica of the service using the new configuration file. Although our fault tolerance and stateless service replication model allows us to easily achieve this restart, this process of stopping individual replicas manually to roll out new changes will be time-consuming. If we were to reimplement this system, we would create an automatic discovery service instead, and the configuration file would only need to store locations of replicas of the discovery service.

Testing

We used two levels of testing to test our system: testing in a local environment and testing in a staging environment. To make our code testable, we made our services take a "sandbox" flag; if sandbox is true, then the current instance of the service will only make local connections to databases, and all required "remote" services will be instantiated locally and provided to our service via a constructor (thus, we do rudimentary dependency injection). To make our tests repeatable, we scripted our Makefile so that when we run "make test", a fresh MongoDB instance will be instantiated locally, all the JUnit unit tests will run, and on success, we terminate the MongoDB instance and clean up any files left behind.

To test in a staging environment, we set up three servers on virtual machines in the SEAS cloud, each server running a replica of every service. In this staging environment, we primarily tested service discovery and fault tolerance. Unfortunately, automating deployment and testing in a distributed environment was sufficiently difficult that we only ran tests manually. For example, we tested our service discovery by manually initiating grading workflows through our system (submitting an assignment, sharding an assignment, submitting a grade for an assignment, checking grades for an assignment), and checking to see if all the services could properly communicate with one another. To test our fault tolerance strategy, we incrementally killed replicas of each service and checked if the service remained up and running.

We instrumented code coverage testing using Emma, an open source Java code coverage tool. When we run "make test", it generates a "coverage" folder containing web pages documenting our code coverage, showing specifically which lines did and did not get executed during our tests. As an example of a coverage report, see our appendix section. As we finalized our development work, we used our Emma coverage reports to identify areas of our codebase that needed further testing. We decided it would be impractical to write programmatic unit tests for our client code; in an ideal world we would look into the logic of the client and programmatically test its interaction with the server, or even instrument browser-level automated interface testing, but we did not do that. Also, because of our "sandbox" style of integration testing, our unit tests do not exercise the remote service lookup functionality found in our services. We also do not have coverage over the main() methods of the services. We tested these components manually, and focused on ensuring that we had code coverage on the methods providing our core application logic.

Conclusion (What we learned)

Lessons we learned

This project gave our team first-hand experience with many of the difficulties inherent in designing distributed systems. It wasn't initially clear to some of us how difficult tasks would become in a distributed setting, even tasks that are quite simple in non-distributed programs, such as generating unique identifiers. We quickly saw that the root of the difficulty in many of these cases was shared persistent state. When two replicas of a component contained state that could become inconsistent, it would be very difficult to prevent and handle these inconsistencies. Our solution to this problem, as was described, was to contain the persistent state to small components and make the central services stateless.

Other solutions we developed during this project used unique properties of this particular application. For example, we assume that a tuple of a student ID, assignment ID, and timestamp uniquely identifies a submission, thus eliminating the need to automatically generate UUIDs, since student IDs and assignment IDs are generated externally to the system. This assumption will generally be valid since we cannot envision a situation in which a student submits two meaningfully different versions of an assignment at the same time. If a student does have two submissions with the same timestamp, we assume they are duplicates. These assumptions may not be valid in a general distributed system. However, in our experience from the distributed systems we have studied this semester, it is necessary in designing any distributed system to make certain assumptions, and the assumptions that are valid will depend on the exact situation.

In addition, we learned that configuring and deploying the system is just as hard as programming the system. Because we used configuration files, we needed to know the topology of the entire system before we launched all the replicas of the services. Furthermore, we had no automated deployment infrastructure, so we had to manually install all dependencies and deploy all replicas via command line over ssh on all the servers, which was very time consuming.

As we built our system, we discovered that making our code testable in both a local and a distributed environment was very difficult. As stated in our TESTING section above, we had to figure out how to make tests automatable, repeatable, and clean up after themselves. Thus, we came up with the "sandbox" strategy to make our services simulate a local environment for testing purposes. In the future, we hope to use a mix of sandboxing and test mocking to write more robust and testable code for local testing. Testing in a real distributed environment, however, was too difficult that we could only do small one-off tests manually.

We also ran into some practical problems on the edge of version compatibility -- the Emma jar file that we are using seems to require Java SDK 1.6 (while the SEAS environment only has Java SDK 1.7). Thus, our unit test suite ("make test") does not work on a VM in the SEAS cloud, and so we have a special ("make seastest") that does not use Emma code coverage. We also experienced some unit test failures that we believe we isolated to MongoDB consistency issues (writing to a database and then nearly immediately reading from it, using a different database connection). So, it is safe to say that we learned our share of practical lessons about the difficulty of getting the nuts and bolts of a system to fit together.

A final lesson was the difficulty of coordinating a team of seven developers with very different schedules. To overcome this difficulty, each member of our team developed an area of expertise. For example, different people were responsible for the frontend, each of the backend services, testing, discovery, etc. Since a large amount of communication between components is necessary, we often needed to coordinate with team members who chose different areas of expertise, but this reduced the amount of whole-group coordination. This system also meant that each component had one team member making sure the component was complete and functional, and recruiting other team members as necessary to help if necessary.

Future Work

For future work, we would like to implement an automatic discovery service, which would allow us to more easily use stateless services that could do more complex computations. For example, by maintaining state across the replicas, we can more easily generate universal unique identifiers (UUIDs), instead of what we currently do (rely on external factors to the system to make assignments and student IDs be unique). The discovery service would keep a mapping between services and the primary replica (or all replicas for state machine replication) for that service. The discovery service itself would be replicated using a primary-backup approach. In a configuration file, we store all currently running replicas of the discovery service. When we start a replica of any service, the replica looks through the configuration file and connects with any replica of the discovery service; the discovery replica responds with the hostname and port of the primary replica of the discovery service. The new replica entering the system then registers itself with the primary discovery replica to allow for other services to communicate with this new replica. Services wishing to communicate with other services query the discovery service to find the hostnames and ports containing replicas of the recipient service.

In addition to this discovery service, we hope to build better infrastructure for testing in a distributed fashion. As stated above, we manually tested our system in the staging environment because automating repeatable tests was too difficult. In the future, we want to research into how real-world distributed systems are tested and incorporate these strategies into our test suite.

Finally, we would like to aggregate statistics and analyze student submissions and grades. We would like to implement machine learning algorithms in the grade compiler service to analyze graders and normalize the distribution of grades; that way, grades given by "hard-ass" graders and more lenient graders will be normalized. Machine learning can also provide administrators with the appropriate weights to assign to graders (e.g. students who have received higher grades are better scorers). In addition, we would like to implement pre-submission hooks into the submission receiver service. For example, in computer science courses, we can automatically run tests on submissions and reject any submissions that failed the tests. Furthermore, we could use pre-submission hooks to notify students of successful submissions and keep track of any late days that students may use.

Appendix & References

Repository

Out git repository for our source code is here: https://github.com/kennyyu/cs262project

Documentation

See our doc/ directory for Javadoc documentation. See our README on how to run our services.

References

Java RMI

- http://docs.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html
- http://docs.oracle.com/javase/1.5.0/docs/api/java/rmi/package-summary.html
- http://docs.oracle.com/javase/tutorial/rmi/running.html

Java Serialization

- http://www.oracle.com/technetwork/java/javase/tech/serializationfaq-jsp-136699.html
- http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

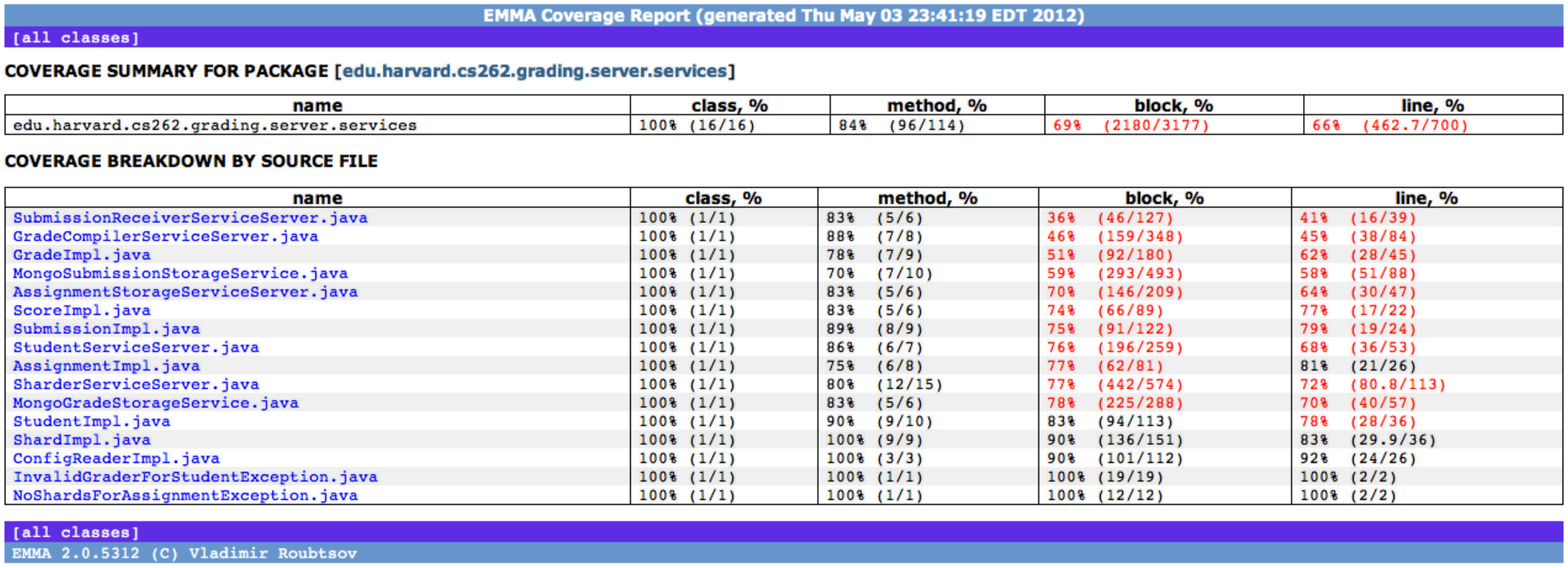
MongoDB

- http://www.mongodb.org/
- http://www.mongodb.org/display/DOCS/Replication

Emma (code coverage)

- http://emma.sourceforge.net/

Emma Test Coverage Report



EMMA Coverage Report (generated Thu May 03 23:41:19 EDT 2012)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [edu.harvard.cs262.grading.server.services]

name	class, %	method, %	block, %	line, %
edu.harvard.cs262.grading.server.services	100% (16/16)	84% (96/114)	69% (2180/3177)	66% (462.7/700)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
SubmissionReceiverServiceServer.java	100% (1/1)	83% (5/6)	36% (46/127)	41% (16/39)
GradeCompilerServiceServer.java	100% (1/1)	88% (7/8)	46% (159/348)	45% (38/84)
GradeImpl.java	100% (1/1)	78% (7/9)	51% (92/180)	62% (28/45)
MongoSubmissionStorageService.java	100% (1/1)	70% (7/10)	59% (293/493)	58% (51/88)
AssignmentStorageServiceServer.java	100% (1/1)	83% (5/6)	70% (146/209)	64% (30/47)
ScoreImpl.java	100% (1/1)	83% (5/6)	74% (66/89)	77% (17/22)
SubmissionImpl.java	100% (1/1)	89% (8/9)	75% (91/122)	79% (19/24)
StudentServiceServer.java	100% (1/1)	86% (6/7)	76% (196/259)	68% (36/53)
AssignmentImpl.java	100% (1/1)	75% (6/8)	77% (62/81)	81% (21/26)
SharderServiceServer.java	100% (1/1)	80% (12/15)	77% (442/574)	72% (80.8/113)
MongoGradeStorageService.java	100% (1/1)	83% (5/6)	78% (225/288)	70% (40/57)
StudentImpl.java	100% (1/1)	90% (9/10)	83% (94/113)	78% (28/36)
ShardImpl.java	100% (1/1)	100% (9/9)	90% (136/151)	83% (29.9/36)
ConfigReaderImpl.java	100% (1/1)	100% (3/3)	90% (101/112)	92% (24/26)
InvalidGraderForStudentException.java	100% (1/1)	100% (1/1)	100% (19/19)	100% (2/2)
NoShardsForAssignmentException.java	100% (1/1)	100% (1/1)	100% (12/12)	100% (2/2)

[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

This is a nice project, and a good writeup. Interesting (but not surprising) that you mirrored the
service-oriented nature of the design with the implementation. It would also be interesting to think about where you could plug other mechanisms in the workflow; one example would be some kind of plagiarism check somewhere in the flow.

Nice job though. I'll have comments on the code and documentation in the future.