Scheduling Tutors in TIPPERS

CS237 Spring 2019
Group #9
Zuozhi Wang
Sadeem Alsudais
Yiheng Xu

1. INTRODUCTION

Our objective is to build a tutor reserving application utilizing TIPPERS. The proposed users of our application are students and tutors. The data sources of our application is the TIPPERS project, which provides recent locations of students and tutors, and our tutors database, which stores the availability and skills of the tutors.
Our application has to meet many requirements. Firstly, a typical use case of our application is that during the final week, with an increased workload, we need to make our application **scalable** by handling users' requests from multiple servers, using load-balancers to distribute the requests. Secondly, we want our application to be **fault-tolerant**, without any single point of failure. Therefore, we should use replication to provide redundancy to our tutors database, in addition to already having multiple request servers. Thirdly, our application must be easy to use, and therefore our scheduling process should be both **fair** and **interactive**. We should ask any tutor if they want to accept a tutoring task, and the asking process should not cause significant delay to the student users.
The requirements listed above lead to the overall design of the project. We aim to use a load balancer to distribute requests among multiple request servers. The request servers will connect to tutors database and TIPPERS database to retrieve the availability and geolocation of both students and tutors. The request servers will advise which tutors are considered best for each tutoring request, and ask them to accept such request. If the request cannot be fulfilled, tutors that are considered "not as good" will be prompted to accept the request. This process will continue to expand tutor search range if no tutors are found or no tutors accepts, until the candidate tutors become "unacceptably bad". During this scheduling process, a student will receive real-time progress updates he can view, and any tutors prompted to accept will receive real-time push notifications. A push notification server will be responsible to send the notifications to tutors given that tutors have subscribed and registered their wish to receive such notifications. We also propose to eliminate race conditions by using mutual exclusion mechanism based on message passing to protect the tutors database and hence, ensuring consistency among different middleware instances.

2. RELATED EFFORTS

In our survey paper we looked into several aspects of our project, including load balancing, replication, stream data processing and publisher/subscriber architecture. In addition to that in our survey paper, we also researched existing solutions for mutual exclusion.

2.1. Load Balancing

According to the paper “Dynamic Load Balancing on Web-Server Systems”[1], there are 4 main types of load balancing. Client-based load balancing, which uses programs on clients to do load balancing, without centralized control and with large granularity. DNS-based load balancing exploits existing DNS records, but has limited control and effectiveness due to intermediate DNS resolvers. Dispatcher-based load balancing grants centralized control and is transparent, but creates bottleneck and single point of failure. Server-based load balancing is distributed, but has increased communication overhead and adds delay to the system.

2.2. Replication

The paper “Replica consistency of CORBA objects in partitionable distributed systems”[2] provides insight into the different forms of replication and the means to keep replicas consistent in the CORBA framework. As we also need to deal with consistency between servers/databases in our project, we use this paper as a reference.

In CORBA, active replication broadcasts all inputs to all same replicated objects. All the replicated objects then perform the same state transition. When returning result, CORBA uses broadcast between replicas to reduce duplicate responses. Passive replication in CORBA broadcast all inputs to one primary, active copy. This primary copy executes state transitions and send the transitions to many cold standbys.

In our survey, we referred to another paper “Comparison of database replication techniques based on total order broadcast”[3]. This paper specifically discusses the replication of databases, and its implementations are based on transactions rather than abstract input to objects in CORBA.

2.3. Stream Data Processing

“The 8 requirements of real-time stream processing” [4] mentions eight requirements to process real time data; which is to keep the data moving; use SQL on streams to query; accept and handle stream imperfections; generate predictable outcomes; integrate stored and streaming data; generate data safety and availability; partition and scale data automatically; finally process and respond instantaneously, which can be done with the help of buffering and scheduling algorithm mentioned in [5].

2.4. Publisher/Subscriber Architecture

“The Many Faces of Publish/Subscribe” [6] discusses the different types of subscriptions such as topic, where the message contains a particular keyword; content, where the message is published when it matches some filters and constraints; finally, type based where it groups commonalities of both topic and content based despite them having different types.

2.5. Mutual Exclusion

The paper “A Simple taxonomy for distributed mutual exclusion algorithms”[7] gives an overview on two categories of mutual exclusion algorithms.

Permission-based algorithms require each process wanting to enter critical session to receive permission from all other processes. If two processes want to enter critical session at the same time, the conflict is resolved between them using some priority mechanism both agrees. For example, in Ricart-Agrawala’s first algorithm, this priority is defined using a logical clock[7].

Token-based algorithms require each process wanting to enter critical session to hold a unique object called a token. The rule for a server to send token takes care of fairness in a token-based algorithm. For example, in Ricart-Agrawala's second algorithm, the most prioritized process to get the token is the next process in an abstract ring.

3. ARCHITECTURE AND DESIGN

In this section we explain the rationale behind some of the design decisions to meet the design goals and system features, and the technical details of how the implementation was done in each component.

1. Load Balancing
    - Design Goal: add scalability to our application by distributing the requests to multiple servers.
    - Implementation Detail: After surveying the existing load balancing solutions, we decided to use the dispatcher-based load balancing. This type of load balancing is transparent to both web client and the server, therefore adds minimal design complexity to both the front-end and the request server.
      Among many dispatcher-based load balancing solutions, we choose NGINX as our load balancer. NGINX [9] is a high performance load balancer, with support for different load-balancing policies, websocket redirection, and automatic avoidance of failed upstream servers. In our setup, client connects to the NGINX server's public IP address. NGINX transparently redirects the messages to and from the back-end servers, by rewriting HTTP headers or IP headers.
2. Stream Data Processing
   We planned to implement stream data processing because we thought we will process data streams from TIPPERS virtual sensors. However, TIPPERS will only grant us access to normal HTTP APIs request. As a result, we cannot implement and have no need for stream data processing.
3. Mutual Exclusion
    - Design Goal: protect the tutors database from race conditions caused by concurrent access from multiple requests and updates, possibly from multiple request servers.
    - Implementation Details: We implemented a simplified version of Ricart-Agrawala's second algorithm using REST API. Our algorithm passes one token among multiple processes, and whichever process has the token can be granted access to read or write the tutors database. Compared to Ricart Agrawala's second algorithm, our token passing rule is local FIFO rather than based on logic clocks. This simplifies the design and reduces delay in message passing.
4. Replication
    - Design Goal: in the event of a failed tutor's database, our application should continue to work properly and should not lose tutors' user data.
    - Implementation Detail: we deployed three instances of postgresql database. When requesting from the tutors database, consensus from all three instances are used.
5. Scheduling
    - Design Goal: to guarantee fairness and liveness when scheduling a tutor, as well as allowing minimal interaction to not disrupt the users.
    - Implementation Detail: the algorithm has multiple rounds. First, retrieve the available tutors with the required skills. Then in each round after that, we apply a filter based on the distance between the student and the tutor to get the top candidates who are closest to the requester. Another filter applied in each round along with the distance is the priority of each tutor. We set the priority of the tutor in a similar fashion of setting the priority of

a thread in an operating system. Each successful reservation causes the priority of a tutor to decrease, and idleness causes priority to increase. In the end, the elected list of tutors will receive push notifications sent via our pub/sub component as described below. Each tutor has to confirm the notification in order to be reserved. If multiple tutors try to confirm at the same time, the first tutor confirms the reservation will be reserved. Each round will relax the distance and priority constraints if the previous rounds were not successful in finding a tutor.

6. Pub/Sub
   Design Goal: for asynchronous and interactive connection, the tutor subscribes to the application that whenever someone is requesting a tutoring session, and he is available and has the requested skill, then he would get a notification of the request from user.
   Implementation Detail: When the tutor is logged in, a web socket connection is established to connect to the push notification component (publisher). Whenever a message of type "reserve" is published, the tutor would be notified and act accordingly if he accepts the request or not. Publisher also passes a variable which contains the requester(middleware's instance information) to let the tutor directly reply accept to the requesting instance(middleware). After the tutor accepts the reservation, the tutor and student can chat directly using websocket connections established between each of them and the middleware server. The tutor has the choice of when to end the tutoring session.

7. Overall Architecture

We use dropwizard framework[10], a Java development framework for RESTful Web Services, to build our middleware and publisher. The majority of the APIs are based on HTTP. The connection between clients and publisher and the connection between student clients and middleware are using websocket.
The below diagram shows the overall system architecture and the integration between the different components.

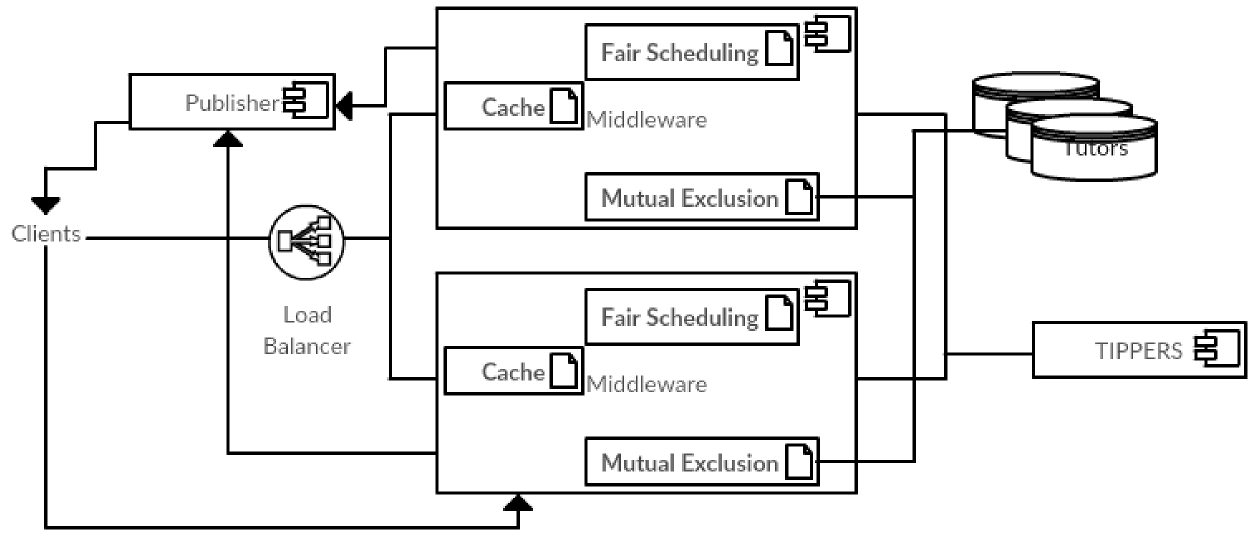


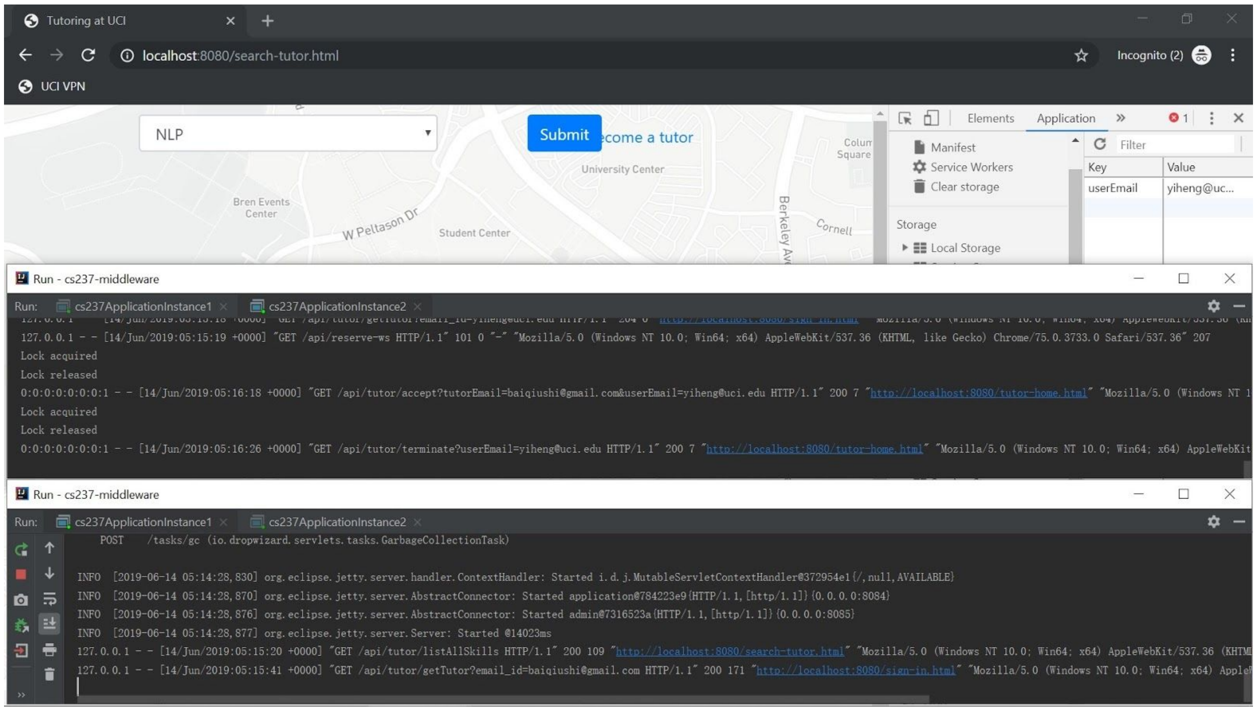
Figure 1: Overall application architecture

4. EVALUATION PLAN AND RESULTS

In this section, we present an evaluation plan for each component and show the experiment results.

The components of our projects include the load balancer, the request server, the push server and the mutual exclusion algorithm. We will test if their basic functionality is met; then, we will evaluate if they meet the design goals of this project.

1. Load Balancer
   - <u>Evaluation Plan:</u>
     We test our NGINX load balancer setup by sending multiple requests to the load balancer's IP address and port. We run two instances of our middleware request server and read the console log. Observing the console windows and the browser tabs will tell us if our load balancer is working correctly.
   - <u>Evaluation Result:</u>
     The result in the graph below shows us the load balancer can distribute request between the load balanced servers. A possible implementation issue is that we want one browser session of one user to stay connected to one request server, so that servers don't need to transfer user sessions; this issue is resolved by keeping each users' session info inside one websocket connection.



```
127.0.0.1 - - [14/Jun/2019:05:15:18 +0000] "GET /api/tutor/getTutor?email_id=yiheng@uci.edu HTTP/1.1" 204 0 "http://localhost:8080/sign-in.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KH
127.0.0.1 - - [14/Jun/2019:05:15:19 +0000] "GET /api/reserve-ws HTTP/1.1" 101 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3733.0 Safari/537.36" 207
Lock acquired
Lock released
0:0:0:0:0:0:0:1 - - [14/Jun/2019:05:16:18 +0000] "GET /api/tutor/accept?tutorEmail=baiqiushi@gmail.com&userEmail=yiheng@uci.edu HTTP/1.1" 200 7 "http://localhost:8080/tutor-home.html" "Mozilla/5.0 (Windows NT 1
Lock acquired
Lock released
0:0:0:0:0:0:0:1 - - [14/Jun/2019:05:16:26 +0000] "GET /api/tutor/terminate?userEmail=yiheng@uci.edu HTTP/1.1" 200 7 "http://localhost:8080/tutor-home.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
```

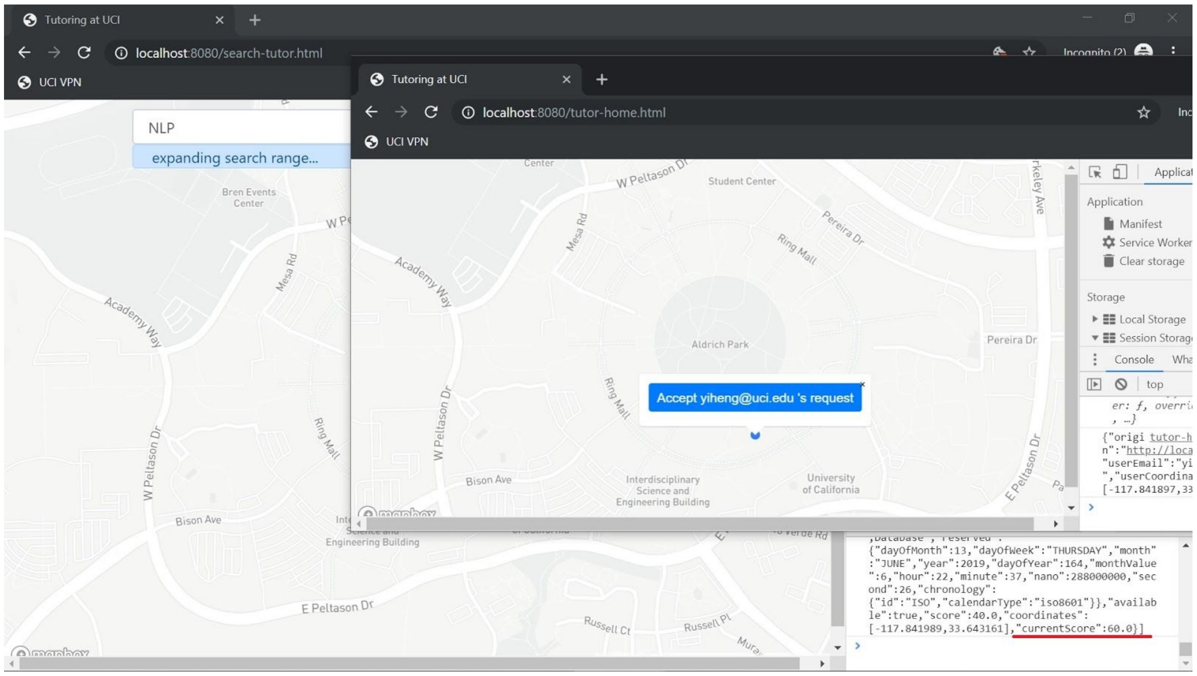
```
    POST    /tasks/gc (io.dropwizard.servlets.tasks.GarbageCollectionTask)

INFO  [2019-06-14 05:14:28,830] org.eclipse.jetty.server.handler.ContextHandler: Started i.d.j.MutableServletContextHandler@372954e1{/,null,AVAILABLE}
INFO  [2019-06-14 05:14:28,870] org.eclipse.jetty.server.AbstractConnector: Started application@784223e9{HTTP/1.1,[http/1.1]}{0.0.0.0:8084}
INFO  [2019-06-14 05:14:28,876] org.eclipse.jetty.server.AbstractConnector: Started admin@7316523a{HTTP/1.1,[http/1.1]}{0.0.0.0:8085}
INFO  [2019-06-14 05:14:28,877] org.eclipse.jetty.server.Server: Started @14023ms
127.0.0.1 - - [14/Jun/2019:05:15:20 +0000] "GET /api/tutor/listAllSkills HTTP/1.1" 200 109 "http://localhost:8080/search-tutor.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
127.0.0.1 - - [14/Jun/2019:05:15:41 +0000] "GET /api/tutor/getTutor?email_id=baiqiushi@gmail.com HTTP/1.1" 200 171 "http://localhost:8080/sign-in.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) Apple
```

Figure 2: Load balancer test result. Two console windows are combined into one screenshot.

2. Middleware request server
   - <u>Evaluation Plan:</u>
     In addition to testing all the APIs in the request server, we supply arbitrary data to the request server. The arbitrary data supplied includes tutors with different availability and locations. It also includes tutors with low search priority and rare skill.
     Then we search for a tutor with a rare skill to see if it can achieve fairness and liveness by expanding search scope of tutors.
   - <u>Evaluation Result:</u>

Our result shows that in the initial two rounds, the tutor of low priority is not selected to receive notification. In the last round, that tutor receives notification and can accept a student's tutoring request. Therefore, the scheduling algorithm can achieve fairness and liveness by expanding search scope of tutors in each students' search tutor request. We can improve our algorithm's effectiveness by adding more rounds to reduce the granularity of each round.


Tutoring at UCI
localhost:8080/search-tutor.html
UCI VPN
NLP
expanding search range...
localhost:8080/tutor-home.html
Accept yiheng@uci.edu 's request
Application
Manifest
Service Worker
Clear storage
Storage
Local Storage
Session Storage
Console

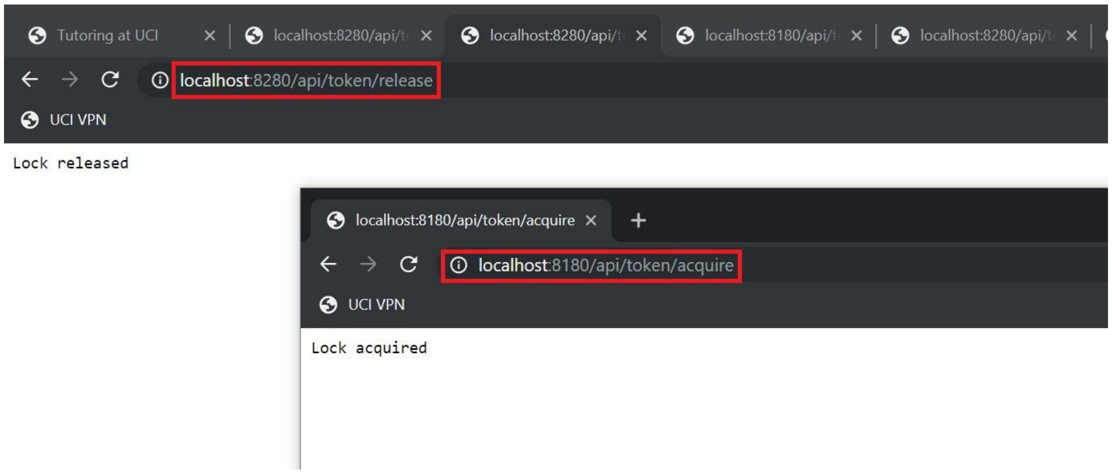
```
{"origi tutor-h
n":"http://loca
"userEmail":"yi
","userCoordina
[-117.841897,33
```

```
{"dayOfMonth":13,"dayOfWeek":"THURSDAY","month"
:"JUNE","year":2019,"dayOfYear":164,"monthValue
":6,"hour":22,"minute":37,"nano":288000000,"sec
ond":26,"chronology":
{"id":"ISO","calendarType":"iso8601"}},"availab
le":true,"score":40.0,"coordinates":
[-117.841989,33.643161],"currentScore":60.0}]
```

Figure 3: Scheduling algorithm test result. After expanding search range, a tutor with a rare skill but with low priority is selected as a candidate. The red line shows the priority of the tutor; higher number means lower priority.

3. Mutual Exclusion Algorithm

- Evaluation Plan:
  Run 2 instances of our mutual exclusion algorithms, inform each of them of others' presence, and create one token in one instance as initialization. Send multiple requests to those servers to acquire permission to enter the critical session. Check that at any moment, permission is only given to one and only one request.
- Evaluation Result:
  We cannot create a race condition requesting permission to enter the critical session. When there is a critical session running, new requests wait acquiring a semaphore with a timeout; a random request gets the permission when current critical session exited.



```
Lock released
```

```
Lock acquired
```

Figure 4: shows different middleware instances (different ports), acquiring after releasing the token.

4. Pub/Sub

- Evaluation Plan:
  We used low level web socket extension for dropWizard [8]. This low level library gives us fine grain control needed to implement topic based pub/sub and saving client sessions in the middleware for later use. To test our finalized pub/sub architecture, log in as a student and try to reserve a tutor. At the same time, log in as a tutor that would be notified. Check if the tutor gets timely push notification.
- Evaluation Result:
  For the final evaluation, as Figure 3 has shown, tutor gets near instant push notification when each round of searching begins.

5. CHALLENGES

- Framework
  At first, we proposed to use the TIBCO framework to implement our middleware. Due to its obscure GUI and the lack of proper documentation, we had to abandon TIBCO and use dropWizard instead.
- User Session Consistency Between Servers
  Another challenge comes from maintaining consistency between multiple middleware servers. Each user's request to search for a tutor creates a session on the middleware server; if load balancer sends the next request to a different server, the user's session info need to be transferred to that server as well. Instead of enforcing ip-hash load balancing policy in NGINX, we keep a websocket connection for each search tutor operation, and use that connection to receive all messages from client side. This solves the consistency issue as well as brings the benefit of progress update to a searching student. If the connected server fails, the student can try again while automatically directed to another server by NGINX.
- Fairness and Liveness
  The main challenge in the scheduling algorithm is to keep balance between fairness and liveness. Fairness can be achieved by giving each tutor a priority, and filtering those with low priority; however, in the event that no tutors with high priority respond, student may fail to find a tutor because low priority tutors never get the notification. To ensure liveness in this scenario, we

divide the searching process to several rounds, and relax the constraints like distance and priority in each round. In this way, Low priority tutors get notified later than those with high priority, but may still get reserved in case they are needed.

6. CONCLUSION AND FUTURE WORK

We have created a tutor scheduling application using TIPPERS that meets our goals in scalability, fault-tolerance, fairness and user interaction. The requests to our application are distributed across multiple middleware servers. For each request to reserve a tutor, the user session is stored in only one middleware server, and this solves the consistency problem between different servers. Our implementation has fault-tolerant tutors database, protected against race condition. Our publisher/subscriber architecture allows timely push notifications to tutors and students alike.
There remains two "single points of failure" in our current implementation: the load balancer and the push server. In future work, we may add redundancy to load balancers using DNS-based load balancing, by resolving domain names of our site to multiple load balancers. This approach will eliminate the load balancer from possible single points of failure. We may also add load balancing and redundancy to the push server, while keeping the one push server abstraction.
Our simplified mutual exclusion algorithm works given the assumption that no critical session hangs. However, if a critical session hangs, the token will never be released. We may implement leader election algorithm to deal with the possible failure where the token is never released due to a server failure.

7. REFERENCES

1. V. Cardellini; M. Colajanni; P. S. Yu, "Dynamic Load Balancing on Web-Server Systems", https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=769420
2. P Narasimhan et al, "Replica consistency of CORBA objects in partitionable distributed systems", https://iopscience.iop.org/article/10.1088/0967-1846/4/3/003/pdf
3. Matthias Wiesmann and André Schiper, "Comparison of database replication techniques based on total order broadcast", https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1401893
4. Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. SIGMOD Rec. 34, 4 (December 2005), 42-47. https://dl.acm.org/citation.cfm?doid=1107499.1107504
5. Zbynˇek Falt and Jakub Yaghob. 2011. Task Scheduling in Data Stream Processing. Dateso pp. 85–96. http://ceur-ws.org/Vol-706/paper18.pdf
6. P.Th. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The Many Faces of Publish/Subscribe. http://members.unine.ch/pascal.felber/publications/CS-03.pdf
7. Michel Raynal. A Simple taxonomy for distributed mutual exclusion algorithms. [Research Report] RR-1362, INRIA. 1991. Inria-00075198. https://hal.inria.fr/inria-00075198/document
8. LivePerson, Inc. "DropWizard Websocket Support", 2017 https://github.com/LivePersonInc/dropwizard-websockets
9. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/
10. A Dallas, "RESTful web services with Dropwizard", 2014 https://dl.acm.org/citation.cfm?id=2636765