02148 - Introduction to Coordination in Distributed Applications

Christian Bach, s134852 Christian Toftemann Bæk, s134867
Lars Kofod Jensen, s134838 Mikkel Møller Brusen, s134568
Sami Ghali, s144443

January 2016

Technical University of Denmark

Statistics management system for the game Hearthstone

Contents

1 Introduction

For at lot of user based applications, it is convenient to have a space in the cloud with all data being processed continuously as it is being received. This is more user friendly than having the user download and process it on their own device. It also allows users to easily share access to a lot of different cloud applications and benefit from their services.

The game Hearthstone is a cross platform online card game with millions of users, that features multiple playable characters. You play the game by choosing a character and a card deck. Then you will be matched with an opponent who likewise has picked a character and card deck. You battle your opponent and the match will end with a win or loss.

In our project we have made a website application for users to submit Hearthstone match reports and see statistics based on those reports. On this website a user can see general statistics, his own statistics and the statistics of his friends. With this application, users can see how they fare with the different characters and make adjustments accordingly. We use Dropbox for our tuple space and this is where all data, submitted from the users, will be send to. Once data has been send to our tuple space, the information will be distributed and processed in order to make statistics and manage friend requests. In this report, we will describe the design of our tuple space structure and how our application coordinate the different processes via multi-threading. For one specific process, we will show how multiple instances of the process can coordinate on different servers. Additionally, we have looked at making the statistics available at all times, even while being updated. In the end, further development will be discussed, in order to make the application more complete.

Link to our website:
`https://mikkelbrusen.github.io/`
`https://mikkelbrusen.github.io/stats/`

Accessing the website and submitting data will require access to the Dropbox folder.

2 Problem Definition

The application will contain different users, user specific statistics, general statistics, and a way to add users to ones friend list so one can see that users personal statistics. In order to do this we will

- Create a tuple space structure
- Feed/read data to/from the tuple space through a website
- Create different processes to sort and calculate data in the different tuple spaces
- Make the processes coordinate via multi-threading

This will be used as a guide line for the project development.

3 Analysis

This section contains an analysis of the program requirements described in the problem definition.

We will be using Dropbox to simulate the different tuple spaces as folders, and the tuples as files. To understand the core principle of tuple spaces and implement it, Dropbox is an obvious choice as the data is accessible through the cloud on all devices. Furthermore we can utilize the basic features from the Dropbox API to create our own API. Since the Dropbox API might not be ideal for this application if it were to scale, we will make sure to design our application in a manner, where transitioning to other technologies would be easy.

We will make a website to create the tuples and we will have multiple processes fetching the tuples, distributing them to the different tuple spaces, and processing the data. The data will then be put as a tuple into a final tuple space and be read by the website. The data read by the website should be as complete as possible. With this, the computations done by the website will be very few, which makes it faster to display the data.

For performance, we want to use multiple threads working on the different tuple spaces, both to sort the files and to create new files with statistics data. Furthermore we'll make it possible to run the processes on multiple systems concurrently using tuples as semaphores to ensure mutual exclusion.

We chose Java as our back-end language since it's easy to use threads and we have extensive experience in the language. Furthermore the use of the Dropbox API is well documented for Java, which greatly improves the workflow. This is also the main reason to use Dropbox, since we do not have to worry about databases and servers, as this is all handled by Dropbox. For the website, JavaScript was an obvious choice since it's easy to learn and known as the language of the web.

4 Design

We started the design phase of the application by discussing the general application structure. In this section we will describe this process.

4.1 Application & tuple space structure

Dropbox is used as our tuple space. A client reads and writes to the tuple space through the front-end. All data from the front-end is placed in the feed folder and is handled by processes running on a series of systems. The processes move files and perform calculations. The overall application structure can be seen in the following figure.

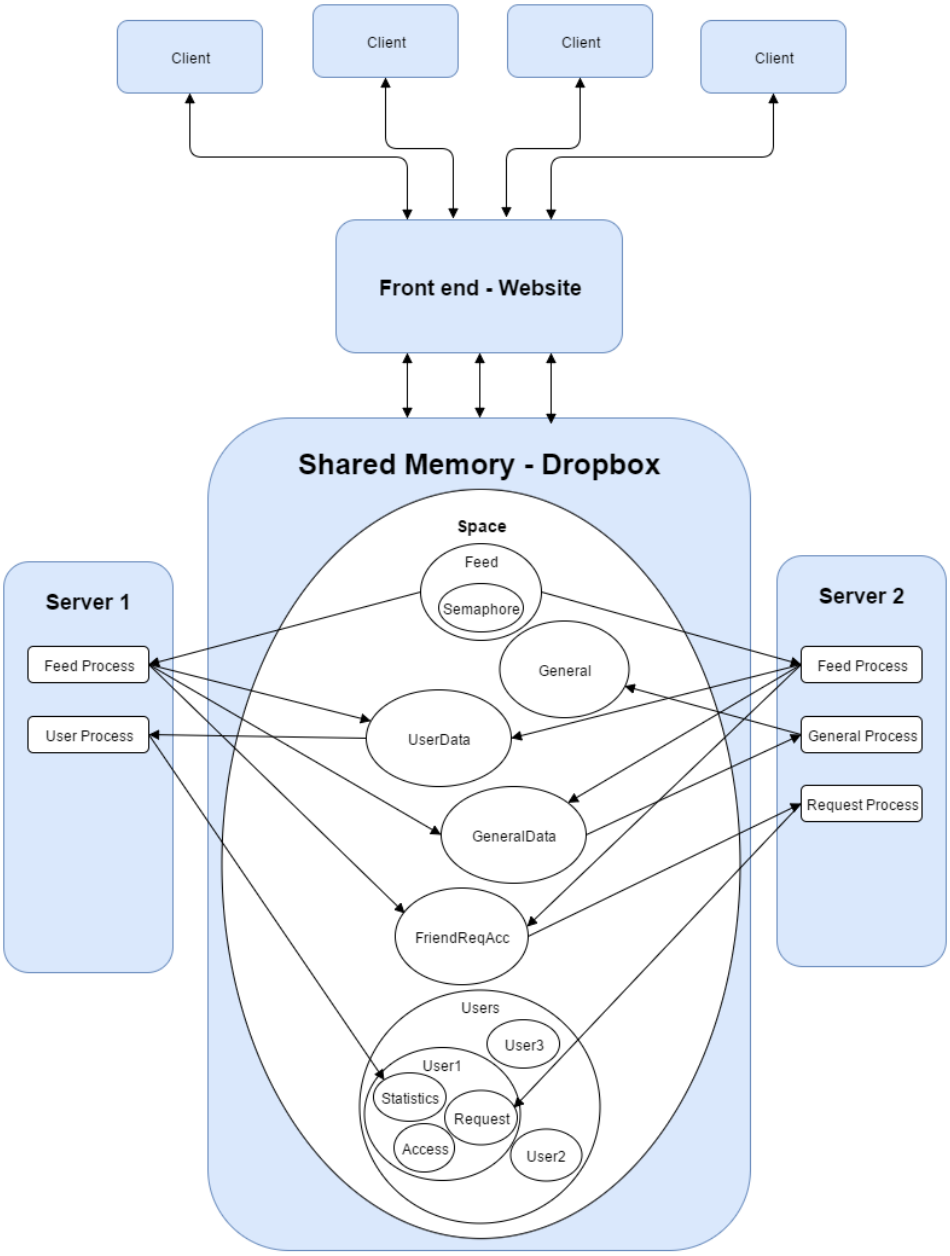

Figure 1: Overall appliction struture describing how different parts of the system interact with each other.

All processes can be run on multiple threads on the same system. Furthermore the feed process can run on multiple systems simultaneously. Ideally, every process should be able to run on multiple systems simultaneously, but we chose to only make this implementation for the feed process. This is because get and put are slow since they use the Dropbox API, but we wanted to show how it could be implemented. The folder structure in Dropbox is used to represent the tuple spaces and their underlying subspaces. The Dropbox folder structure can be seen in the following figure.

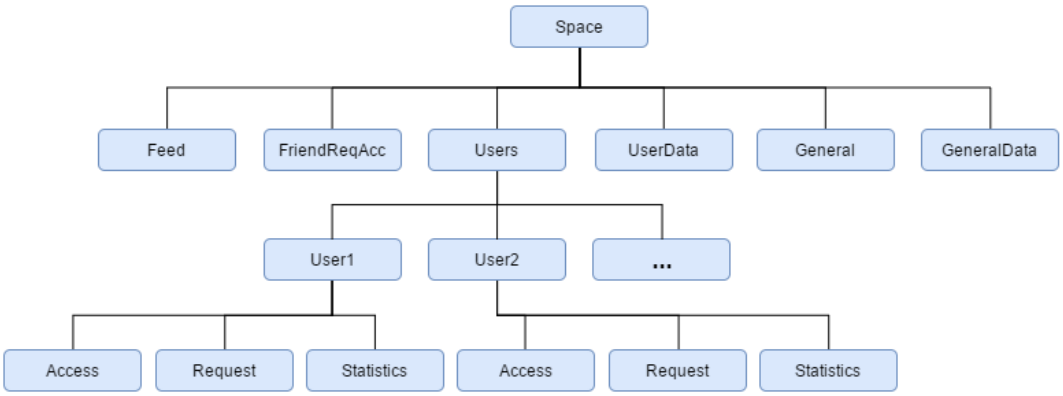

Figure 2: Dropbox folder structure

The `Feed` folder is where input from the front-end is placed. We have a designated feed process to sort tuples from the feed folder into `FriendReqAcc`, `UserData`, and `GeneralData`. We then have designated processes to handle computations on tuples in these folders, which overall improves the performance, since the feed process doesn't need to process the data.

In `UserData` we have a process that checks for new files appearing in the folder. These files will be analysed and used to update the statistics for the specific user. The files in the `GeneralData` folder are being handled by a different process. This process updates the general statistics, reflecting the performance of all the users in our tuple space. Lastly we have a process monitoring the `FriendReqAcc` folder. This process checks for new friend requests being sent between users and sorts them out into user specific `Request` folders. If two users have requested each other, the process will update the users `Access` folder, allowing them to see each others user statistics.

Because we use files to represent our tuples and we wish to allow multiple instances of the same tuple, we use uniquely generated file extensions. This is done to avoid having files called `tuple_info.12344`, `tuple_info (1).12344` and so on. We designed our filenames and thereby our tuples to look the following way:

Tuple description		
Tuple represents:	**Filename:**	**Notes:**
A match:	USER_MYCLASS_OPPCLASS_RESULT	Where the result is represented as either "Win" or "Loss"
Statistics for a class:	CLASS_WINS_LOSSES	The win ratio can then be calculated on the client side.
A friend request:	REQ_USER1_USER2	Here USER2 represents the user sending a friend request to USER1.
Request or access to statistics:	USER	Here USER is placed in the folder of the user he has access to or requested

Figure 3: Description of the naming scheme for the different tuples.

From these tuples it is possible to extract all data needed. The file name of a match includes OPPCLASS. This is the class of the opponent. In our current application we do not use it, but it could be used if we wanted to add statistics for each class match-up.

4.2 API

We found it necessary to extend the Dropbox API. This provides us with the functionality needed for the operations performed by our application. Thus our API works as a shell that accesses the Dropbox API, which then again accesses the data in the dropbox folder space.

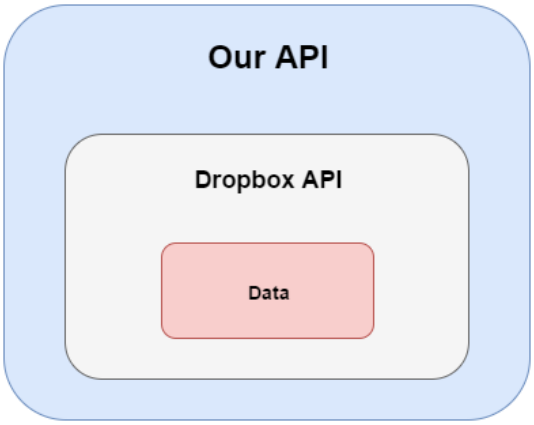

Figure 4: Describes the connection between our API and the one provided to access Dropbox data.

Our API is used by each of the processes to modify the data in Dropbox. Our API uses the functionality provided by the Dropbox API to construct a new set of primitives. These will be described in a later section.

4.3 Frontend

We've made a website in order to feed tuples to our tuple spaces and get an easy overview of the statistics made by our Java back-end. The website consists of two pages, one for feeding the tuples, and one for viewing the statistics. In order to communicate with Dropbox we use an unofficial JavaScript library for the Dropbox Core API called dropbox.js[1].

Below we've included some screenshots from the frontend. In the first image on the left, you choose which character you were playing. In this case a mage is selected. On the right side you pick your opponents character. Then you input your username and select whether you won or lost. When clicking submit, a file is sent to Dropbox with the selected information.

On the second image, our statistics page is shown. Here you are able to fetch your own statistics, the general statistics, and other users statistics, given that you are friends with the other user. You can also see requests sent to you, as well as accepting/sending requests to other users.

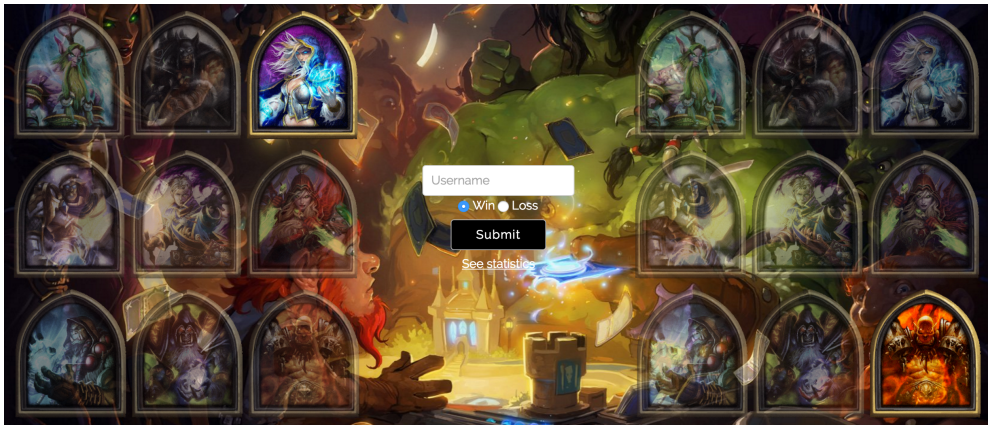

Figure 5: Front page

Figure 6: Statistics page

[1] `https://github.com/dropbox/dropbox-js`

5 Implementation

In this section we will describe the thought process of the implementation throughout the development of the application.

5.1 System processes

Here, the job of the four processes monitoring the folders `Feed`, `GeneralData`, `UserData` and `FriendReqAcc` will be described. Pattern matching is used in all processes to ensure that files are being correctly distributed and updated.

With the specific design of our input files, we can split the filenames according to underscores. In the feed process, by looking at the first split item, it will determine where to redirect the input file. If the file starts with "REQ", then it's a friend request and is being send to the `FriendReqAcc` folder. If the file starts with a username, then it's a match report and it will be send to the folders `GeneralData` and `UserData`. We found it necessary to have a feed folder, in order to distribute the tuples to different processes. Doing this, we avoid having multiple different types of processes interfering with each other, in the same folder.

First of, the process in `GeneralData` handles a match report by discarding the username. Then it searches the `General` folder for a file matching the second split item, containing the class the user played as. If there exists a matching file, then there has already been recorded some statistics for the specific class. In this case we update the statistic file according to the result of the input file. Otherwise we just create a new file for the class, with the corresponding win or loss.

In the `UserData` folder, the process uses pattern matching in a similar way as the process in `GeneralData`. But instead of discarding the username of the input file, it uses the username to search the `Users/username` folder for an existing file matching the class used by the user. It then continues to update the statistics file for the specific user.

The process monitoring the `FriendReqAcc` folder uses the `USER1` and `USER2` information to see if it appears in one of the users `Request` and `Access` folders. If one of the users are already represented in the other users `Access` folder, then the friend request file is discarded. If `USER2` has `USER1` in his `Request` folder, then this implies a mutual consent for friendship and both users will be put in each others `Access` folder. To ensure that both users are actual users, existing in our tuple space, we check to see if both users have a statistics folder in a matching username folder. This is because when a person submits his first match report, he instantly becomes a user and gets a statistics folder.

5.2 Primitives

We have a class `Template` from which we do all operations and pattern-matching on the tuple space. The `Template` has a name and a path, and contains the primitives of our API layer. The path describes which tuple space we operate in, and the name describes the tuple(s) we are searching for or have found. We have extended the Dropbox API so we have the following primitives:

- **Get:** Works like `out`. It reads a tuple from the tuple space, and deletes it afterwards.

- **Remove:** Deletes the file on a certain path. Used if a file has to be read and deleted in two steps.
- **ExistsWithPrefix:** Searches for a tuple with a specific prefix and reads the tuple. Returns a boolean depending on tuples existence. It is used for updating class stats.
- **Put:** Works like `in`. It puts a file into the tuple space.
- **DoesExist:** It tries to read a specific tuple in the user tuple space, and return a boolean depending on the existence of this tuple. It is used to check if a user exists.
- **SearchFor:** It tries to read a tuple on a given path. It returns a boolean depending on the existence of this tuple. It is used to check if users have requested each other.

The `put` primitive uses Dropbox' `uploadFile`. This function requires a byte input describing the content of the file to be uploaded. This cannot be `null`. Therefore, we add a '1' byte to all files we want to upload in order for this function to work. This is an inelegant solution, but necessary nonetheless.

As mentioned, `Template` contains all the API calls. Therefore, if we choose to use another technology than Dropbox, it is basically only this class, which needs changing. Of course there are some other parts of the program, for instance containing specific Dropbox paths, which needs to be changed as well. However, these changes are few and quite simple.

5.3 System coordination

The application uses semaphores. The semaphores are used to make certain operations atomic. We have implemented our processes so they extend `Thread`. We do this because we want more than one instance of a process to run at a time. Each of the processes have at least one static semaphore. The semaphore is implemented in two different ways.

1. The first semaphore implementation has a counter, which is decreased when `P` is called and increased when `V` is called. `P` and `V` are synchronized, so only one thread can access them at a time. If the counter is 0 when `P` is called, the thread has to wait. After each `V` operation the waiting threads are notified.
2. The second version is a file located on Dropbox. When an atomic action is wanted, the process uses `get` on the file. The process then uses `put` on the file once the action is done.

The first implementation of the semaphore only works locally. Therefore, when it is used, the process can run with multiple of the same process threads, but only on the same system. The second semaphore version is located on the tuple space. This makes it possible for multiple systems to coordinate the actions for multiple of the same process threads. This would be the optimal version if `get` and `put` were not as slow as they currently are. We therefore only use the second implementation in the feed process, to show that it could be done. In the remaining processes we use the first implementation.

When a process uses `get` on a file, it does not know the name of the file in advance. Therefore, the `get` operation has to be atomic. Otherwise two different versions of the same process might try to get the same file. This will cause an exception in one of the threads, since only one of the processes will be able to delete the file, and will cause it to crash. This could be optimized if we had an algorithm, which correctly could sort out the

tuples, so no two processes would ever try to get the same tuple.

It is only the `get` operation, which is atomic, for the most part of the application. This means that the run time of a series of the same process is determined by the run time of `get`. When the first process has run `get`, the next process can run `get`. The optimal number of threads for the same process can, therefore, be calculated by:

- Run time for `get`: x.
- Maximum run time for *the entire process*: z.
- Then z can be written as $z = x \cdot y$ where $0 < x < z$ and $0 < y < z$

Here y will be the optimal number of threads given by $y = z/x$, assuming there always are available tuples. However, this does not work for `ReqAccFeed`, since the same semaphore is used two times in a single run loop.

5.4 Front-end

We use JavaScript as our middle-end in order to bind our tuples to the front-end. This way, the client does not have access to delete files. It only has access to write to the feed space, and read from stats, request and access spaces.

We check if a user has access to read another users info by first reading the logged in users access folder. If a tuple with a key matches that of the user he is requesting to read statistics from, the statistics are shown. By doing it like this, we load the statistics in one cycle, rather than having the front-end wait for the back-end to make the checks. Not very secure, but fast.

In order to put a tuple into the feed space, we use writeFile from dropbox.js library as shown here:

```
var user = $('#user').val();
var myClass = $('input[name="myClass"]:checked').val();
var theirClass = $('input[name="theirClass"]:checked').val();
var result = $('input[name="result"]:checked').val();
var id = Math.random().toString(36).substr(2, 5);

...

client.writeFile("/space/Feed/"+user+"_"+myClass+"_"+
    theirClass+"_"+result+"."+id, "", function(error, stat)
```

Figure 7: Dropbox write file

The user has full control of what should be sent, as long as all input fields are filled. This means that a user can commit false data, but we chose not to make more advanced validation system for this relatively small application.

5.5 Tuple availability when reading

A feature we found important in our application, was to ensure availability of our statistics, to the user. To start of with, we had issues when reading the statistics tuple while updating them, because we simply took down the file and put up a new updated version. In that time the file was inaccessible and the statistics page would show the statistics as "N/A".

Before we came to our final solution, we considered having a log file, used for reading, for every statistics file in our system. This would mean having both a read and write file for each statistics file in our tuple space. So when a user requests some statistics, the front-end would read the log file. If a user submits a match report, the system would overwrite the write file. This would mean twice as many statistics files in our tuple space, which would scale a lot, when the application takes in new users. However, in this context, we would still need to synchronize the log file with the write file. This could be done as a batch job, once a day, when the user activity is low. The downside to this implementation is, that the statistics would not be completely up to date and at some point of the day, the statistics would be unavailable. Nevertheless, the statistics will be available as long as the batch job is not running.

Our final solution ended up being a quite simple way of changing our initial protocol of updating the files. When writing, we now read the statistics file, put up the updated version of it and only then remove the old statistics file. In this case, the statistics will always be available to the users. In some cases, an old and a new file version, of some statistics, will exist. We found two main issues with this. The first one being, that a second process should never intervene with the first process, by trying to update the old statistics file, before it has been removed. This we solved by making the new protocol atomic for each class statistic file, forcing a thread process to grab a semaphore before executing the protocol. The second issue is when both statistics files are present, and the front-end might choose the old version. We chose to oversee this, as it doesn't really affect much in a system where the deviation would be unnoticeable.

6 Discussion

This section includes concerns about the current program version, and describe how this can be optimized by further development.

The files in the metadata of a Dropbox folder is sorted alphabetically. The get operation takes the first file found in the metadata. Therefore, if the workload is large, a file beginning with 'Z' might wait a long time to be processed. This is bad because the program should not differentiate between users depending on their username. A better version would be to queue the files in the folders depending on the time they were created. This will make the processing of the files more fair. Another solution would be to develop an algorithm which sorts out the tuples so the get operation does not have to be atomic. An example could be to have different versions of the same tuple space for each of the processes working on the tuple space. The algorithm has to make sure that the number of files in each of these tuple spaces are distributed equally. This would decrease the process time of a tuple and thereby solve the problem.

An issue with our system, is when we terminate our processes. When coordinating processes from different servers, we use a semaphore uploaded to our tuple space. Terminating the processes may lead to a semaphore disappearing, if it is currently occupied by a thread. As of yet, this only concerns the feed process, but ideally, all types of processes should be able to run simultaneously on different servers. We have no way of properly restarting the application, without re-uploading the semaphores manually. Another similar issue would be terminating a process that is updating statistics. If the process gets terminated before finishing its protocol, there might be multiple versions of specific statistics files in our tuple space. This will lead to a loss of data. A way to solve this problem, is by having another protocol for terminating processes. This protocol should send out some type of turnstile signal, so the processes will finish their current job and stop processing more data. Only when all processes are waiting at the turnstile, the application should be terminated. However, this doesn't take into account a system crash from a power failure. This would require some sort of system recovery protocol.

The match report contains an opponent class, which were intended for more advanced statistics. But since this would just be a repetition of already used methods, we choose not to, and rather focus on multi-threading on single systems and being able to run the process concurrent on multiple systems. However, it would still be a nice addition in a future application version.

The application uses the Dropbox API. The get and put operations are slow. Naturally, this makes the use of tuple semaphores slow as well. If we used a different, more efficient API the tuple semaphore solution would be better. The program is designed so a change of API would require few changes, mainly only in the Template class. One reason, for the operations being slow, is because data has to be transferred between the Dropbox servers and the computers running the processes. This is done every time a tuple is being processed. Centralising the data and the processes on a single server would improve performance a lot. This would also remove the reliance on the file semaphore used in feed, since all processes would be running on the same system. We could also use something different than a file system to hold our tuples. For example, a database system with a tuple space implementation.

7 Conclusion

During the project we have created a tuple space in Dropbox. Using this tuple space we were able to share the data amongst our different users. The data in the tuple spaces are presented to the user through a website. This website contains the basic functionality such as viewing statistics, adding match info, and sending friend requests. To sort the data in our tuple space we implemented a series of processes in Java. These processes can modify the data and place it in different tuple spaces. The processes are all capable of coordinating via multi-threading. Furthermore we extended a process so it can run on multiple systems simultaneously. Throughout the project we have focused on solving the technically difficult tasks, such as coordinating multiple threads. This is prioritized higher than expanding our program with features that are nice, but just are a repetition of something we have already implemented. Thus we omitted to implement a feature that extended our statistics by allowing the user to view character to character win ratios. Had this been implemented, it would primarily include an extension of the front-end rather than the way we handle tuples. To modify data in our tuple space from the different processes we implemented a series of primitives in our own API. These features functions as a shell surrounding the Dropbox API and allow us to do specific operations. Furthermore, we discuss future development and concerns regarding the application.

All this has allowed us to make an application that lives up to the problem definition.