In [35]:

```
%matplotlib inline
```

Overview

This practical required us to work with a large dataset of tweets using Python and Jupyter, in particular carrying out analysis and visualisation tasks.

Implemented Requirements

Basic Requirements

- Removing duplicates
- Refined data file production
- Automatic data refinement
- Calculating total number of tweets, retweets, and replies
- Calculating the number of different users tweeting in the dataset
- Calculating the average number of tweets, retweets and replies sent by a user
- Finding the most popular hashtags
- Visualisation of activity structure
- Visualisation of tweet activity over time
- Wordcloud visualisation of other hashtags
- Providing Jupyter notebook to re-run analysis

Additional Requirements

- Analysing applications used to send tweets
- Analysing patterns of user activity over time
- Analysing and visualising interactions between users

Problems

The main problem with the practical is its somewhat fragmented structure. It would be a great improvement in terms of reusability, maintainability, and ease of understanding to have, for instance, universal functions to filter retweets from the DataFrame. Although the approach is usually the same, having an abstracted method to do so would also make it easy to implement and debug system functionality.

Other than this, the code has been tested and seems to work and give reasonable outputs: it appears to function correctly.

Few, if any, problems were found in the development of the solution; if any, the learning curve of using specific library functionality is probably the largest obstacle we had to overcome in the system's development. There were no large problems in terms of difficult bugs to solve, requirements that we didn't know how to implement at all, etc.

Reproducibility and Reusability

As previously discussed, the reusability and maintainability of the program could have been greatly increased by using abstracted methods for common actions, such as filtering or reading in data, even though the same or very similar approaches are presently used every time the action occurrs.

Assuming another dataset has exactly the same structure, the program will be reproducilbe; the only possible gaps we could think of this would be when filtering out the specific hashtag #CometLanding. This would mean if tweets relating to another hashtag were being analysed, the code would have to be manually edited to exclude the new hashtag from the top hashtags and other similar functionality.

Provenance

Sources

- studres start code
- 'Fastest way to uniqify a list in Python' (https://www.peterbe.com/plog/uniqifiers-benchmark)

Contributions

taeh

- Implementing descriptive analysis section for basic specification
- Small fixes elsewhere in basic specification to complete it
- Analysis and visualisation of interactions between users
- Writing "Report-like" sections of Jupyter notebook (Overview, Requirements implemented, etc.)
- All code explanations in Jupyter notebook, and controlling its refactoring to include more than calls to .py files
- Refactoring practical as a whole to meet requirements

feb7

- Analysis of applications used to send tweets
- Analysis of user activity over time
- Startpoint for "report-like" section of Jupyter notebook (Overview, Requirements implemented, etc.)

yd23

- The basic specification except for (missing or incorrect implementations - mostly only incorrect) descriptive analysis section and dataset structure visualisation

Analysis of Tweets Containing the CometLanding Hashtag

Preprocessing input CSV data

In [36]:

```
from parsing import *
cr = CensusReader("../data/CometLanding.csv")
```

```
File contains 51 invalid rows!
Refining the data given and storing it a new file...
Refined data can be found in '../data/CometLanding_refined.csv'.
```

CensusReader class __init__() method

The __init__() method in the CensusReader class calls read_csv() from pandas on the file CometLanding.csv.
This creates a DataFrame which is then parsed with validateFile(); invalid tweets are removed in this process.

In [37]:

```
def __init__(self, fileName):
    self.fileName = fileName
    self.data = pd.read_csv(fileName)

    self.data["valid"] = True
    self.validateFile()
```

validateFile() method

The validateFile() method finds all duplicte entries in the DataFrame (by their string tweet ID).
It then removes all the duplicate entries from the DataFrame, before printing the number of duplicate tweets (invalid rows) there were.
Finally, it outputs the remaining (valid, non-duplicate) entries/tweets in the DataFrame to a new CSV file.
It outputs it to the filename the data was read in from, with _refined added before the .csv extension; the data is converted from a DataFrame into CSV format using the to_csv() method.

In [38]:

```
def validateFile(self):
        #check duplication
        self.data["valid"] = self.data["valid"] & ~(self.data.duplicated("id_st
r"))

        invalidData = self.data.query("valid == False")
        self.data = self.data.query("valid == True")

        # Drops the 'valid' column since it is no longer needed.
        self.data.drop('valid', axis=1, inplace=True)

        invalidRows = len(invalidData)

        if (invalidRows > 0):
            if invalidRows == 1:
                print("File contains 1 invalid row!")
            else:
                print("File contains "+str(invalidRows)+ " invalid rows!")

            #make a new file
            if self.fileName[-4:] == ".csv":
                print("Refining the data given and storing it a new file...")
                newFileName = self.fileName[:-4] + "_refined.csv"
                self.data.to_csv(newFileName)
                print("Refined data can be found in \'" + newFileName + "\'.")
```

Counting the numbers of tweets, replies, retweets, and users, and calculating average user activity.</h3>

In [39]:

```
from parsing import *

cr = CensusReader("../data/CometLanding_refined.csv")
retweets = cr.retweetCount()
replies = cr.replyCount()
tweets = cr.tweetCount()
users = cr.userCount()

print("\nThere were " + str(tweets) + " tweets in total.")
print("There were " + str(replies) + " replies in total.")
print("There were " + str(retweets) + " retweets in total.\n")

print("There were " + str(users) + " users tweeting in the dataset.\n")

print("The average number of tweets by a user in the dataset was " + str(tweets
/ users) + ".")
print("The average number of replies by a user in the dataset was " + str(replie
s / users) + ".")
print("The average number of retweets by a user in the dataset was " + str(retwe
ets / users) + ".")
```

```
There were 15547 tweets in total.
There were 1723 replies in total.
There were 59998 retweets in total.

There were 50192 users tweeting in the dataset.

The average number of tweets by a user in the dataset was 0.30975055
7858.
The average number of replies by a user in the dataset was 0.0343281
797896.
The average number of retweets by a user in the dataset was 1.195369
7800446286.
```

retweetCount() method

The retweetCount() method first finds if the tweet text for each item in the DataFrame from the CSV data starts with "RT", indicating it is a retweet.
Each result is added to a new list of Boolean values; the sum() method is then used to count the number of True values in the list (i.e. retweets).
The value of the sum function is then returned.

In [40]:

```
def retweetCount(self):
    return (self.data['text'].str.startswith("RT")).sum()
```

replyCount() method

The replyCount() method first finds if the 'in_reply_to_screen_name' field for each item in the DataFrame from the CSV isn't empty, indicating it is a reply.
Each result is added to a new list of Boolean values; the sum() method is then used to count the number of True values in the list (i.e. replies).
The value of the sum function is then returned.

In [41]:

```
def replyCount(self):
    return (self.data['in_reply_to_screen_name'].notnull()).sum()
```

tweetCount() method

The tweetCount() method first finds the number of unique tweets. It does this by taking all parsable entries in the DataFrame (from which duplicates have already been dropped).
It then finds the length of the resulting DataFrame (of all unique tweets), representing the total number of tweets of any kind.
To then find the number of tweets that are not replies or retweets, we then subtract the result of the replyCount() and retweetCount() methods from the number of unique tweets found.

In [42]:

```
def tweetCount(self):
    return len(self.data['entities_str'].notnull()) - self.replyCount() - self.r
etweetCount()
```

userCount() method

The userCount() method first finds entries in the DataFrane who have unique posting users (i.e. removing all but one tweet posted by each user with one or more tweet in the dataset)
It does this with the pandas drop_duplicates() method.
The length of the resulting DataFrame is then returned, representing the number of unique uesers who posted tweets in the dataset.

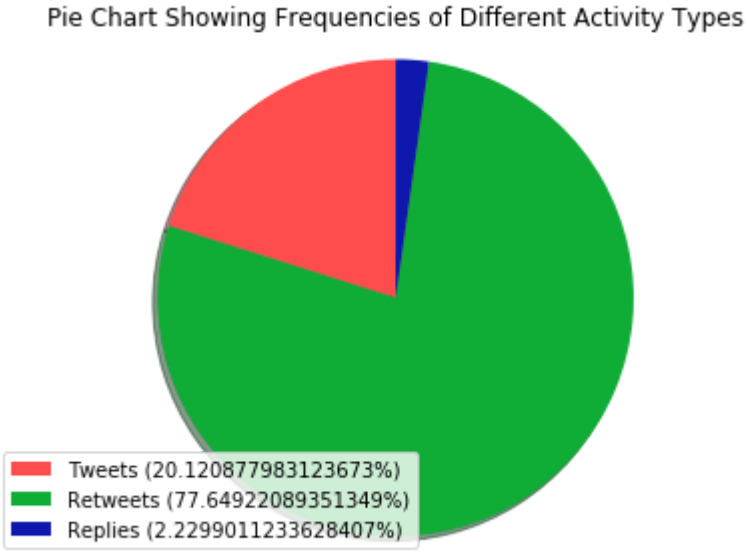
In [43]:

```
def userCount(self):
    return len(self.data.drop_duplicates(subset=['from_user_id_str']))
```

Pie chart showing composition of activity by type

In [44]:

```
from parsing import *
from plotter import *

cr = CensusReader("../data/CometLanding_refined.csv")

retweets = cr.retweetCount()
replies = cr.replyCount()
tweets = cr.tweetCount()

plotter = Plotter()
plotter.pieChart(tweets, retweets, replies)
```



pieChart() method

The pieChart() method begins by adding the numbers of tweets, retweets, and replies in the DataFrame.
It then finds the percentage proportion of all activity for each activity type (tweets, retweets, and replies).
It then creates lists of labels for the legend, the "slice" sizes, and colours of the pie chart.
These and other arguments are then passed to plt.pie() and other plt() methods to generate and then display the pie chart (plt represents matplotlib.pyplot).

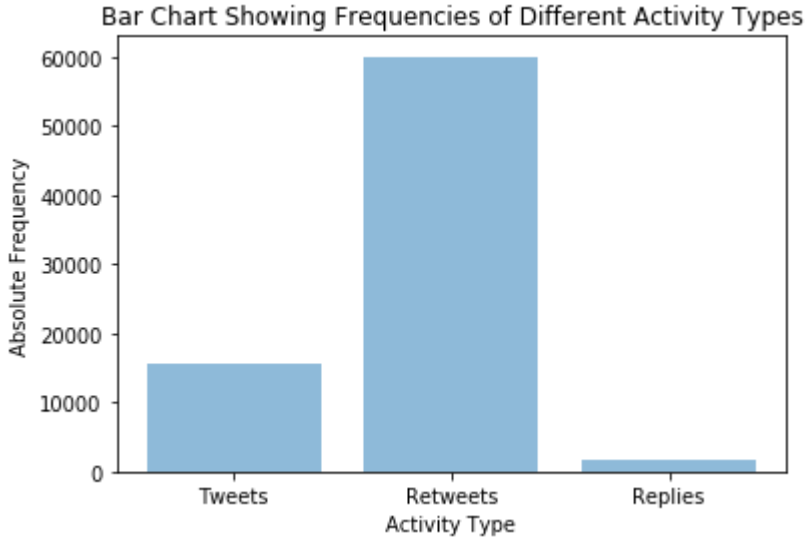
In [45]:

```
def pieChart(self, tweets, retweets, replies):
    total = tweets + retweets + replies
    # The slices will be ordered and plotted counter-clockwise.
    percents = [(float(tweets)/float(total)*100.0), (float(retweets)/float(total
)*100.0),(float(replies)/float(total)*100.0)]

    labels = 'Tweets (' + str(percents[0]) + '%)', 'Retweets (' + str(percents[1
]) + '%)', 'Replies (' + str(percents[2]) + '%)'
    fracs = [tweets, retweets, replies]
    colors = ['#ff4d4d', '#0FAC36', '#0F18AC'] # red, green, blue

    patches, texts = plt.pie(fracs, colors=colors, shadow=True, startangle=90)
    plt.legend(patches, labels, loc="best")
    plt.axis('equal')

    plt.tight_layout()
    plt.title('Pie Chart Showing Frequencies of Different Activity Types')
    plt.show()
```

Bar chart showing composition of activity by type

In [46]:

```
from parsing import *
from plotter import *

cr = CensusReader("../data/CometLanding_refined.csv")

retweets = cr.retweetCount()
replies = cr.replyCount()
tweets = cr.tweetCount()


plotter = Plotter()
plotter.barChart(tweets, retweets, replies)
```



The barChart() method

The barChart() method is very similar to the pieChart() method, using the numbers of tweets, retweets, and replies in the DataFrame.
Again, values to be used as parameters for the bar chart are generated from the fequency of each activity type; these are used with their Names, as well as axis labels and a title to generate the bar chart, again with plt as matplotlib.pyplot.

In [47]:

```
def barChart(self, tweets, retweets, replies):
    vals = [tweets, retweets, replies]
    labels = ('Tweets', 'Retweets', 'Replies')
    n_groups = len(vals)
    bar_width = 1/1.5
    pos = np.arange(len(labels))

    plt.bar(pos, vals, align='center', alpha=0.5)
    plt.xticks(pos, labels)
    plt.ylabel('Absolute Frequency')
    plt.xlabel('Activity Type')
    plt.title('Bar Chart Showing Frequencies of Different Activity Types')

    plt.show()
```

Finding the 10 most common hashtags used

In [48]:

```
from parsing import *

cr = CensusReader("../data/CometLanding_refined.csv")
print(cr.mostPopHashtags(10, False))
print("\n\n")
print(cr.mostPopHashtags(10, True))
```

```
1. #CometLanding : 62752 tweets
2. #cometlanding : 13211 tweets
3. #67P : 7922 tweets
4. #Rosetta : 5912 tweets
5. #Philae : 3097 tweets
6. #Cometlanding : 1036 tweets
7. #WishKoSaPasko : 988 tweets
8. #HappyBirthdaySandaraPark : 965 tweets
9. #rosetta : 702 tweets
10. #esa : 680 tweets



1. #67P : 7922 tweets
2. #Rosetta : 5912 tweets
3. #Philae : 3097 tweets
4. #WishKoSaPasko : 988 tweets
5. #HappyBirthdaySandaraPark : 965 tweets
6. #rosetta : 702 tweets
7. #esa : 680 tweets
8. #SEP : 623 tweets
9. #philae : 600 tweets
10. #PoseToiPhilae : 569 tweets
```

The mostPopHashtags() method

The mostPopHashtags() method begins by getting sorted list of hashtags and their numbers of occurances from the getHTagsAndCounts() method.
The method then checks if the user wants to exclude variants of the #CometLanding hashtag from the results; if so, the method will iterate over the items in the list up to the number to be printed, removing any matches to the #CometLanding hashtag and then moving back.
The method then iterates over the resultant list of hashtags up to the nth largest to be printed, printing each hashtag and the number of tweets in the DataFrame it appears in.

In [49]:

```
def mostPopHashtags(self, number, excludeCometLanding):
    sortedHTagsAndCounts = sorted(self.getHTagsAndCounts(), key = itemgetter(1),
 reverse=True)
    returnString = ""

    if excludeCometLanding:
        max = number - 1
        x = 0
        while x < max:
            if sortedHTagsAndCounts[x][0].lower() == "cometlanding":
                del sortedHTagsAndCounts[x]
            else: x += 1

    for x in range(0,number):
        returnString += str(x + 1) + ". #" + sortedHTagsAndCounts[x][0] + " : "
+ str(sortedHTagsAndCounts[x][1]) + " tweets\n"

    return returnString
```

The getHTagsAndCounts() method

The getHTagsAndCounts() method begins by getting a list of every hashtag; each is present in the list the number of times it is tweeted as hashtags are extracted from tweets and added to the list for each tweet. This is done in the getHashTags() method.
This list is then sorted, before a list without duplicates is found using the removeDuplicates() method.
This new list without duplicates is then iterated over; each hashtag in it, and the number of times it is present in the original list of hashtags (found using a Counter) is added in a new array item in a new 2D array, hTagsAndCounts, containing each hashtag, and the number of times it is present in a tweet in the DataFrame.
This new 2D array is then returned.

In [50]:

```
def getHTagsAndCounts(self):
    hashtags = sorted(self.getHashTags(), reverse=False)
    hashtagsNoDuplicates = self.removeDuplicates(hashtags)
    hTagsAndCounts = []
    counter = Counter(hashtags)
    for ht in hashtagsNoDuplicates:
        count = counter[ht] # hashtags.count(ht)
        hTagsAndCounts.append([ht, count])
    return hTagsAndCounts
```

The getHashTags() method

The getHashTags() method generates a list of each hashtag each time it is in a tweet in the DataFrame.
It begins by removing all fields but 'entities_str' for each entry in the DataFrame using the loc pandas function.
It then generates a JSON object from the 'entities_str' field for each entry in the DataFrame using json.load().
The JSON array of hashtags are then extracted from this, and iterated over; each hashtag in the JSON hashtag list in the JSON entities object for each entry (tweet) is also then iterated over. The text of each hashtag is then appended to the list of hashtags being build.
When all iteration terminates, the resultant list of hashtags is returned.

In [51]:

```
def getHashTags(self):
    #     hashtags = self.data.groupby('entities_str').count().sort_values(by=
['id_str'], ascending=False).head(number)
    data = self.data
    hashtags = []
    jsonItems = data.loc[:, 'entities_str'] # lst of JSON hashtag items
    # each JSON hashtag item has syntax: {""hashtags"":[{""text"":""67P"",""indi
ces"":[58,62]},{""text"":""CometWatch"",""indices"":[127,138]},{""text"":""Comet
Landing"",""indices"":[139,140]}],""symbols"":[],""user_mentions"":[{""screen_na
me"":""ESA_Rosetta"",""name"":""ESA Rosetta Mission"",""id"":253536357,""id_st
r"":""253536357"",""indices"":[3,15]}],""urls"":[{""url"":""http://t.co/Z2A14Jor
v6"",""expanded_url"":""http://youtu.be/4a3eY5siRRk"",""display_url"":""youtu.b
e/4a3eY5siRRk"",""indices"":[104,126]}]}".
    for JSONString in jsonItems:
        if not type(JSONString) is str: continue
        parsedJSON = json.loads(JSONString)
        JSONHashTagsElement = parsedJSON['hashtags']
        for JSONHashTag in JSONHashTagsElement:
            hashtags.append(JSONHashTag['text'])
    return hashtags
```

The removeDuplicates() method

Copied from https://www.peterbe.com/plog/uniqifiers-benchmark (https://www.peterbe.com/plog/uniqifiers-benchmark).
This method adds all the items in the list seq to a dictionary as keys; if an identical item is added it will override the previous entry for the identical key, thus eliminating duplicates.
The keyset from the dictionary, copied from the seq list, is then returned.

In [52]:

```
def removeDuplicates(self, seq):
    # Not order preserving
    keys = {}
    for e in seq:
        keys[e] = 1
    return keys.keys()
```

Hashtag wordcloud

Hashtag wordcloud

The wordcloud code first filters all but the JSON entities field from each entry in the DataSet.
Each entities field (for each entry) is then iterated over, and is parsed into a JSON object.
Each hashtag in the hashtags object in the entites JSON object is then iterated over.
If the text (field) of the hashtag is not "CometLanding" in some form, it is added to the list of hashtags being built.
The WordCloud function from the wordcloud library is then used to build up a wordcloud image, and plt (as matplotlib.pyplot) is then used to display the wordcloud.

In [53]:

```
from parsing import *
import json
from wordcloud import WordCloud
import matplotlib.pyplot as plt

data = CensusReader("../data/CometLanding_refined.csv").data
hashTags = data.loc[:, 'entities_str']
words = []
for i, v in hashTags.iteritems():
    try:
        j = json.loads(v)
        for tag in j['hashtags']:
            if tag['text'].lower() != "cometlanding":
                words.append(tag['text'])
    except TypeError:
        pass

wordcloud = WordCloud(width=1000, height=500).generate(" ".join(words))
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

Most common platforms

In [54]:

```
from parsing import *

cr = CensusReader("../data/CometLanding_refined.csv")
print (cr.appUsed())
```

```
Top Platforms used
1. 'Twitter Web Client', 27925
2. 'Twitter for iPhone', 13743
3. 'Twitter for Android', 12770
4. 'TweetDeck', 4075
5. 'Twitter for iPad', 3282
6. 'dlvr.it', 1671
7. 'Twitter for Websites', 1462
8. 'Tweetbot for iÎŸS', 1055
9. 'Twitter for Windows Phone', 932
10. 'Tweet Old Post', 925
11. 'Twitter for Mac', 915
12. 'Twitter for Android Tablets', 897
13. 'Hootsuite', 796
14. 'TweetCaster for Android', 477
15. 'Ø§Ø´ØªØ±ÙƒÙ€Ù€ Ø§Ù„Ø¢Ù†', 356
16. 'Twitter for BlackBerryÂ®', 345
17. 'Mobile Web M5', 315
18. 'IFTTT', 303
19. 'Twitter for BlackBerry', 291
20. 'Tweetbot for Mac', 289
```

appUsed() method

The appUsed() method first filters the DataFrame, keeping only the source field in each entry.
Each source field (in each entry), is added to a dictionary; if it is a new key it sets the number of occurances (value) as 1, otherwise increasing the value by 1.
The resultant dictionary is then sorted by largest to smallest keys (number of occurances).
A for loop then iterates over each key-value pair in the dictionary, generating a human-readable string representing the contents of the dictionary, which is then returned and then printed.

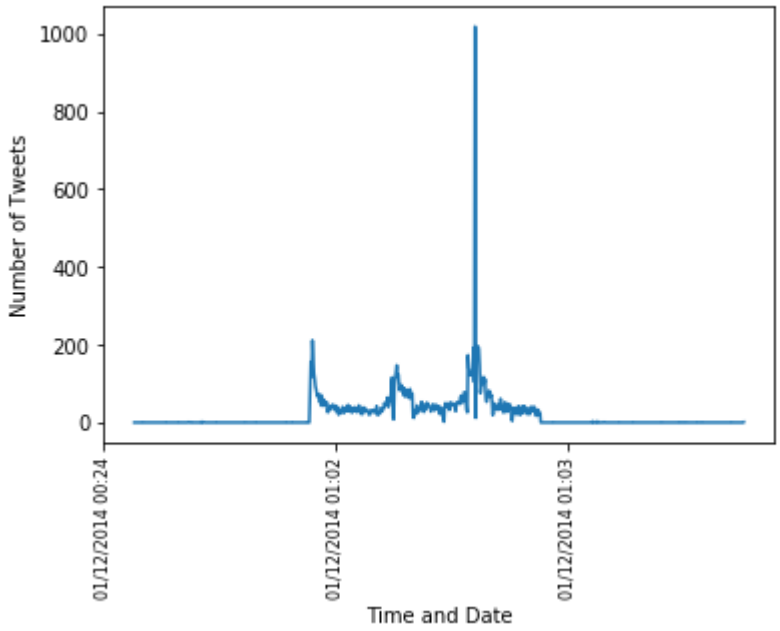
In [55]:

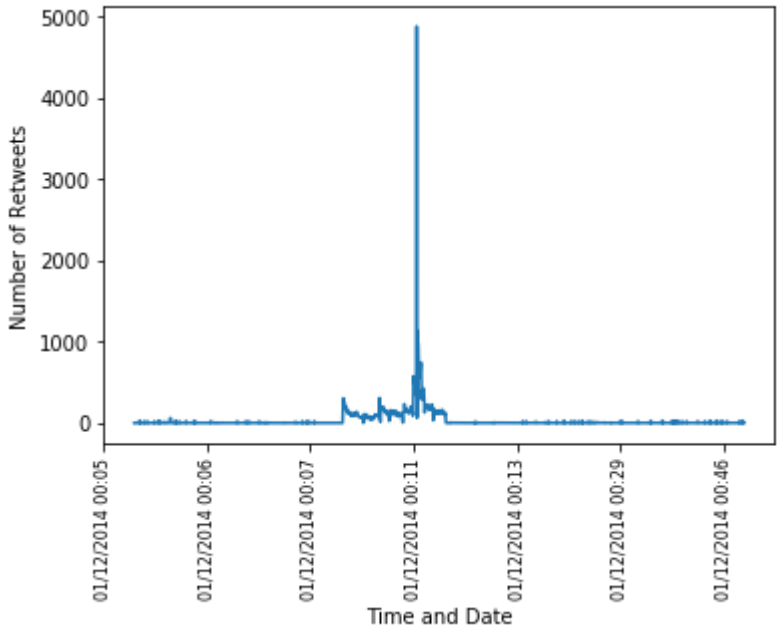
```
def appUsed(self):
    apps = self.data['source'].str.extract("<.*>(.*)</a>", expand = False)
    platforms = {}
    for x in apps:
        y = str(x)
        if y in platforms:
            platforms[y] += 1
        else:
            platforms[y] = 1
    import operator
    sortPlat= sorted(platforms.items(), key=operator.itemgetter(1), reverse=True
)

    returnString = "Top Platforms used \n"
    for m in range(0,20):
        returnString += (str(m + 1) + ". " + str(sortPlat[m]) + "\n").replace("
(", "").replace(")", "")
    return returnString
```

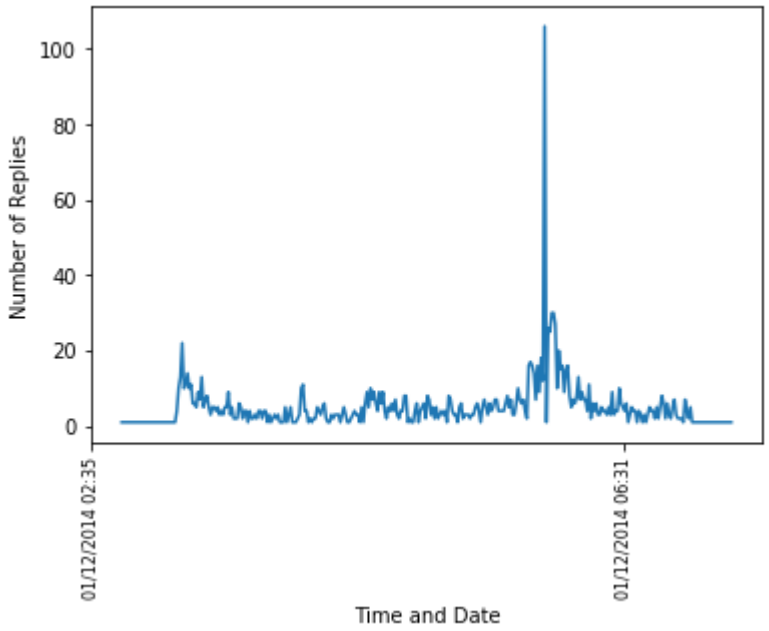
Tweet Activity Timelines

In [56]:

```
from parsing import *
from plotter import *

cr = CensusReader("../data/CometLanding_refined.csv")
data = cr.data

plotter = Plotter()

plotter.tweetsTimeLine(data)
plotter.retweetsTimeLine(data)
plotter.repliesTimeLine(data)
```


Number of Tweets
1000
800
600
400
200
0
01/12/2014 00:24
01/12/2014 01:02
01/12/2014 01:03
Time and Date


Number of Retweets
5000
4000
3000
2000
1000
0
01/12/2014 00:05
01/12/2014 00:06
01/12/2014 00:07
01/12/2014 00:11
01/12/2014 00:13
01/12/2014 00:29
01/12/2014 00:46
Time and Date


Number of Replies
100
80
60
40
20
0
01/12/2014 02:35
01/12/2014 06:31
Time and Date

tweetsTimeLine() method

The tweetsTimeLine() method begins by filtering out any retweets, finding only valid tweets.
It then groups each entry in the DataFrame of valid tweets by the time field, using the pandas groupby() method.
Each tweet time is then iterated over, and the number of tweets at each time is found, and stored in a separate list, num.
These lists are preprocessed and then passed into methods imported from matplotlib.pyplot (as plt) to draw the timeline before it is displayed.

In [57]:

```
def tweetsTimeLine(self, data):
    tweets = data[data['text'].str.startswith("RT") == False]
    groupbytime = tweets.groupby('time').count()
    time = groupbytime.iloc[:, 0]
    count = 0
    num = [] #number of tweets
    xticks = []
    for i, v in time.iteritems():
        count += 1
        num.append(v)
        xticks.append(i)

    y = num
    fig, ax = plt.subplots()
    ax.plot(y)
    start, end = ax.get_xlim()
    ax.xaxis.set_ticks(np.arange(start, end, 300))
    ax.xaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))
    ax.set_xticklabels(xticks, rotation='vertical', fontsize=8)
    ax.set_xlabel('Time and Date')
    ax.set_ylabel('Number of Tweets')
    plt.show()
```

retweetsTimeLine() method

The retweetsTimeLine method is largely identical to tweetsTimeLine().
The differences between the two are axes' labels, and that the tweets used in this case are filtered to be retweets.
This is done by ensuring the text field of each entry in the DataFrame (representing the tweet text) starts with "RT".

In [58]:

```
def retweetsTimeLine(self, data):
    retweets = data[data['text'].str.startswith("RT") == True]
    groupbytime = retweets.groupby('time').count()
    time = groupbytime.iloc[:, 0]
    count = 0
    num = [] #number of tweets
    xticks = []
    for i, v in time.iteritems():
        count += 1
        num.append(v)
        xticks.append(i)

    y = num
    fig, ax = plt.subplots()
    ax.plot(y)
    start, end = ax.get_xlim()
    ax.xaxis.set_ticks(np.arange(start, end, 300))
    ax.xaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))
    ax.set_xticklabels(xticks, rotation='vertical', fontsize=8)
    ax.set_xlabel('Time and Date')
    ax.set_ylabel('Number of Retweets')
    plt.show()
```

repliesTimeLine() method

The repliesTimeLine method is also largely identical to tweetsTimeLine().
The differences between the two are axes' labels, and that the tweets used in this case are filtered to be replies.
This is done by ensuring the text 'in_reply_to_screen_name' of each entry in the DataFrame is not null (i.e. there is someone the tweet is replying to).

In [59]:

```
def repliesTimeLine(self, data):
    replies = data[data['in_reply_to_screen_name'].notnull() == True]
    groupbytime = replies.groupby('time').count()
    time = groupbytime.iloc[:, 0]
    count = 0
    num = [] #number of tweets
    xticks = []
    for i, v in time.iteritems():
        count += 1
        num.append(v)
        xticks.append(i)

    y = num
    fig, ax = plt.subplots()
    ax.plot(y)
    start, end = ax.get_xlim()
    ax.xaxis.set_ticks(np.arange(start, end, 300))
    ax.xaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))
    ax.set_xticklabels(xticks, rotation='vertical', fontsize=8)
    ax.set_xlabel('Time and Date')
    ax.set_ylabel('Number of Replies')
    plt.show()
```

Finding and analysing user interactions

Finding and analysing user interactions

Finding and analysing user interactions

The initial list of replies, retweets, and mentions are found by selecting from the DataFrame.
The replies are found by only keeping entities whose value for the '' field is not null; retweets are found by checking the 'text' field of an entity starts with "RT"; mentions are found by checking that the 'text' field of an entity contains an @ symbol, and also that the requirements for replies and retweets both fail. This ensures retweets and replies are not counted as mentions.
The retweetsToTwoUsers(), repliesToTwoUsers(), and mentionsToTwoUsers() methods are then used to convert the resultant variables retweets, replies, and mentions into 2D arrays of the usernames of both users involved in the interaction.
A resultant list of all interactions is formed by concatenating all 3 formed arrays; the lengths of each list of interactions at each stage of processing are then analysed.

In [60]:

```
from parsing import *
from plotter import *

plotter = Plotter()

print("Finding user interactions.\n")

data = CensusReader("../data/CometLanding_refined.csv").data

replies = data[data['in_reply_to_screen_name'].notnull()]
retweets = data[data['text'].str.startswith("RT") == True]

mentions = data[data['text'].str.contains("@") == True]
mentions =  mentions[mentions['in_reply_to_screen_name'].notnull() == False]
mentions = mentions[mentions['text'].str.startswith("RT") == False]

print("There were " + str(len(replies)) + " replies prior to processing.")
print("There were " + str(len(retweets)) + " retweets prior to processing.")
print("There were " + str(len(mentions)) + " mentions containing mentions prior
 to processing.\n")

formattedRetweets = plotter.retweetsToTwoUsers(retweets)
formattedReplies = plotter.repliesToTwoUsers(replies)
formattedMentions = plotter.mentionsToTwoUsers(mentions)

interactions = np.concatenate((formattedReplies, formattedMentions, formattedRet
weets), axis=0)

retweets = len(formattedRetweets)
replies = len(formattedReplies)
mentions = len(formattedMentions)

print("There were " + str(replies) + " replies after processing.")
print("There were " + str(retweets) + " retweets after processing.")
print("There were " + str(mentions) + " mentions after processing.")
print("Note: a tweet containing >0 mentions can contain >1 mention.\n")

print("There were " + str(len(interactions)) + " interactions in total after pro
cessing.\n\n")
```

```
Finding user interactions.

There were 1723 replies prior to processing.
There were 59998 retweets prior to processing.
There were 5044 mentions containing mentions prior to processing.

There were 1641 replies after processing.
There were 53377 retweets after processing.
There were 5972 mentions after processing.
Note: a tweet containing >0 mentions can contain >1 mention.

There were 60990 interactions in total after processing.
```

retweetsToTwoUsers() method

The retweetsToTwoUsers() method takes in a DataFrame containing retweets, iterating over each entry (tweet).
The user who sent the tweet is found from the 'from_user' field.
The user being retweeted is found by parsing the tweet text, in the 'text' field.
Both usernames are then added to the list of interactions, as a sub-array.
The list of interactions is returned after duplicates are removed from it.

In [61]:

```
def retweetsToTwoUsers(self, data):
    interactions = []
    for index, row in data.iterrows():
        sentBy = row['from_user']
        tweetText = row['text']

        if sentBy == None or tweetText == None or type(tweetText) != str or type
(sentBy) != str: continue
        splitText = tweetText.split()
        if len(splitText) < 2: continue

        retweeting = splitText[1].replace(":", "").replace("@", "")
        interactions.append([sentBy, retweeting])

    return DataFrame(interactions).drop_duplicates()
```

repliesToTwoUsers() method

The repliesToTwoUsers() method is largely the same as retweetsToTwoUsers().
The second user is now the user being replied to, and is found from the 'in_reply_to_screen_name' field.

In [62]:

```
def repliesToTwoUsers(self, data):
    interactions = []
    for index, row in data.iterrows():
        sentBy = row['from_user']
        inReplyTo = row['in_reply_to_screen_name']
        if sentBy == None or inReplyTo == None or type(sentBy) != str or type(in
ReplyTo) != str: continue
        interactions.append([sentBy, inReplyTo])
    return DataFrame(interactions).drop_duplicates()
```

mentionsToTwoUsers() method

The mentionsToTwoUsers() method is largely the same as retweetsToTwoUsers().
There are now potentially more than one second users for each first user, as a tweet sent by a single user can mention multiple other users.
The twitter text, from the 'text' field, is therefore parsed for @ symbols; the username is extracted as the text after each @ symbol and before the next whitespace character.
Each username mentioned is then added to the list of interactions alongside the user who sent the tweet.

In [63]:

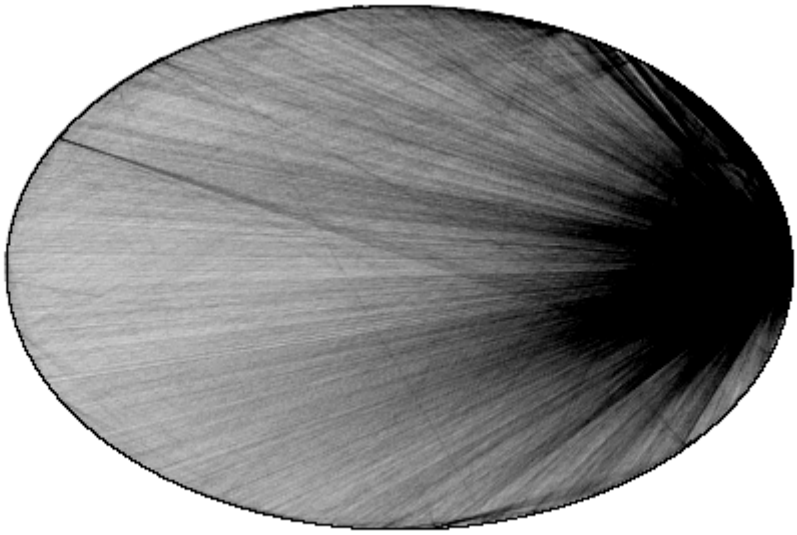
```
def mentionsToTwoUsers(self, data):
    interactions = []
    for index, row in data.iterrows():
        sentBy = row['from_user']
        tweetText = row['text']

        if sentBy == None or tweetText == None or type(tweetText) != str or type
(sentBy) != str: continue
        splitText = tweetText.split()
        if len(splitText) < 1: continue
        else:
            for string in splitText:
                if (string.startswith("@")):
                    interactions.append([sentBy, string])

    return DataFrame(interactions).drop_duplicates()
```

Network graph of all interactions

In [64]:

```
from plotter import *
from parsing import *

plotter = Plotter()

data = CensusReader("../data/CometLanding_refined.csv").data

replies = data[data['in_reply_to_screen_name'].notnull()]
retweets = data[data['text'].str.startswith("RT") == True]

mentions = data[data['text'].str.contains("@") == True]
mentions =  mentions[mentions['in_reply_to_screen_name'].notnull() == False]
mentions = mentions[mentions['text'].str.startswith("RT") == False]

formattedRetweets = plotter.retweetsToTwoUsers(retweets)
formattedReplies = plotter.repliesToTwoUsers(replies)
formattedMentions = plotter.mentionsToTwoUsers(mentions)

interactions = np.concatenate((formattedReplies, formattedMentions, formattedRet
weets), axis=0)

# above code previously explained in the "Finding and analysing user interaction
s" section

plotter = Plotter()
plotter.networkGraph(interactions, "interactions")
print("\nYou can see on the graph the majority of interactions end at the right
 side of the graph, where they converge.")
print("This suggests a small number of users are interacted with many times by o
ther users.")
print("I believe it may be the official account for the comet landing mission tw
eeting about its success.")
print("This is also the case on the following network graphs for retweets, repli
es, and mentions only.")
```

```
Now drawing the graph showing all interactions between users.
```



```
Graph density = 2.5182130445852386e-05

You can see on the graph the majority of interactions end at the rig
ht side of the graph, where they converge.
This suggests a small number of users are interacted with many times
by other users.
I believe it may be the official account for the comet landing missi
on tweeting about its success.
This is also the case on the following network graphs for retweets,
replies, and mentions only.
```

The networkGraph() method

The networkgraph() method uses networkx, imported as nx. It first performs a print statement informing the user of what the graph represents, before generating and displaying the graph.
The list of username pairs in the DataFrame given are iterated over, and an edge is added between each pair.
The width of the lines are adjusted to best display the connections between nodes for particular inputs (found by trial and error).
The parameters are then passed to generate and then display the graph, before some properties of the graph, found with library functions, are printed.

In [65]:

```
def networkGraph(self, plotItemList, whatPlotting):
    G = nx.DiGraph()

    print("Now drawing the graph showing all " + whatPlotting + " between user
s.")

    if (whatPlotting == "interactions"): plotItemList = DataFrame(plotItemList)

    for index, row in plotItemList.iterrows():
        G.add_edge(row[0], row[1])
    UG = G.to_undirected()

    width = 0.003
    if (whatPlotting == "replies"): width = 0.4
    if (whatPlotting == "mentions"): width = 0.07

    options = {
        'node_color': 'black',
        'node_size': 1,
        'line_color': 'grey',
        'linewidths': 0,
        'width': width,
    }

    nx.draw_circular(UG, **options)
    plt.show()
    print("Graph density = " + str(nx.density(G))) # TODO: print some more prope
rties of the graph
    if (whatPlotting != "interactions" and whatPlotting != "mentions"): print("
\n\n")
```

Network Graphs of Retweets, Replies, and Mentions

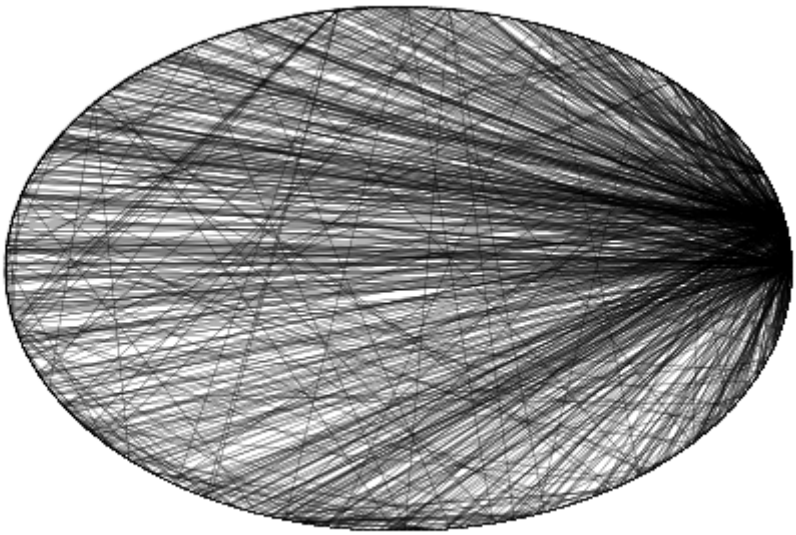
In [66]:

```
from plotter import *
from parsing import *

plotter = Plotter()

data = CensusReader("../data/CometLanding_refined.csv").data

replies = data[data['in_reply_to_screen_name'].notnull()]
retweets = data[data['text'].str.startswith("RT") == True]

mentions = data[data['text'].str.contains("@") == True]
mentions =  mentions[mentions['in_reply_to_screen_name'].notnull() == False]
mentions = mentions[mentions['text'].str.startswith("RT") == False]

formattedRetweets = plotter.retweetsToTwoUsers(retweets)
formattedReplies = plotter.repliesToTwoUsers(replies)
formattedMentions = plotter.mentionsToTwoUsers(mentions)

# for code explanation see "Network graph of all interactions" section

plotter = Plotter()
plotter.networkGraph(formattedRetweets, "retweets")
plotter.networkGraph(formattedReplies, "replies")
plotter.networkGraph(formattedMentions, "mentions")
```

Now drawing the graph showing all retweets between users.

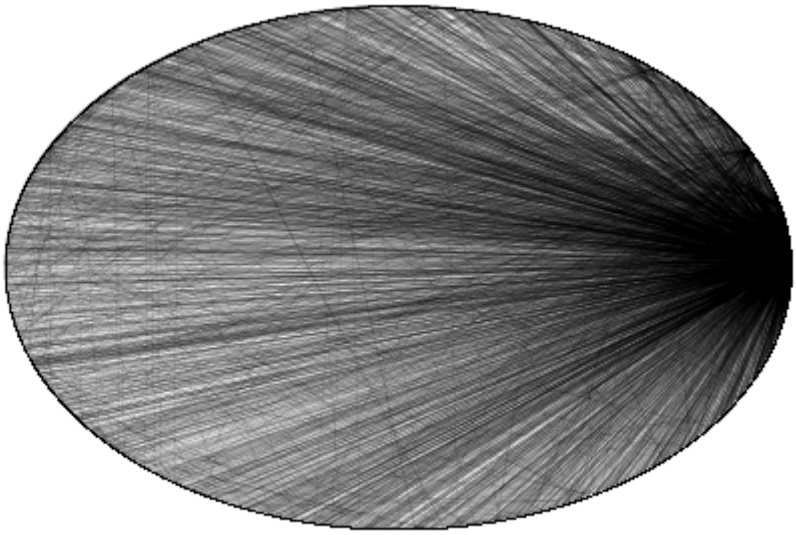
Graph density = 2.692865021425508e-05

Now drawing the graph showing all replies between users.



Graph density = 0.000364948893811214

Now drawing the graph showing all mentions between users.

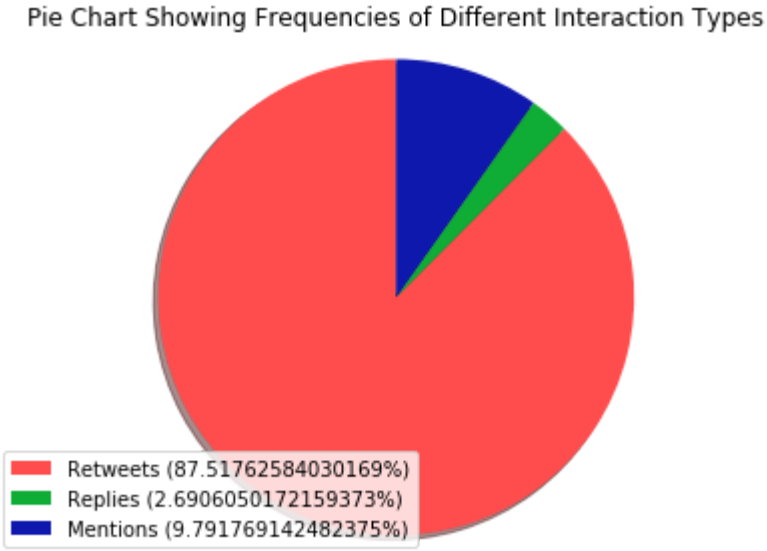


```
Graph density = 0.00021895905282390816
```

Pie Chart Showing Make-up of Interactions By Interaction Type

In [67]:

```
from plotter import *
from parsing import *

plotter = Plotter()

data = CensusReader("../data/CometLanding_refined.csv").data

replies = data[data['in_reply_to_screen_name'].notnull()]
retweets = data[data['text'].str.startswith("RT") == True]

mentions = data[data['text'].str.contains("@") == True]
mentions =  mentions[mentions['in_reply_to_screen_name'].notnull() == False]
mentions = mentions[mentions['text'].str.startswith("RT") == False]

formattedRetweets = plotter.retweetsToTwoUsers(retweets)
formattedReplies = plotter.repliesToTwoUsers(replies)
formattedMentions = plotter.mentionsToTwoUsers(mentions)

# for explanation of above code, see "Finding and analysing user interactions" s
ection

plotter = Plotter()
plotter.interactionsPieChart(len(formattedRetweets), len(formattedReplies), len(
formattedMentions))
```



interactionsPieChart() method

The interactionsPieChart() method takes in the number of retweets, replies, and mentions.
It then calculates the percentage shares of each interaction type, out of all interactions.
It then generates the labels for each type of interaction that include its percentage share of all interactions.
These arguments are then passed into various functions from plt (matplotlib.pyplot) to generate the pie chart, before it is given a title and then displayed.

In [68]:

```
def interactionsPieChart(self, retweets, replies, mentions):
    total = retweets + replies + mentions
    # The slices will be ordered and plotted counter-clockwise.
    percents = [(float(retweets)/float(total)*100.0), (float(replies)/float(tota
l)*100.0),(float(mentions)/float(total)*100.0)]

    labels = 'Retweets (' + str(percents[0]) + '%)', 'Replies (' + str(percents[
1]) + '%)', 'Mentions (' + str(percents[2]) + '%)'
    fracs = [retweets, replies, mentions]
    colors = ['#ff4d4d', '#0FAC36', '#0F18AC'] # red, green, blue

    patches, texts = plt.pie(fracs, colors=colors, shadow=True, startangle=90)
    plt.legend(patches, labels, loc="best")
    plt.axis('equal')

    plt.tight_layout()
    plt.title('Pie Chart Showing Frequencies of Different Interaction Types')
    plt.show()
```