GameBadger: Design and Development of a Social Gaming Platform

Brendan Curran, Ángel Rodríguez

{*brendanmcurran, angelrh*}*@ischool.berkeley.edu*

Advisor: John Chuang

UC Berkeley School of Information

May 2012

Contents

1 Introduction

Over the last half-decade, mobile gaming markets have grown at an exceptional rate: worldwide revenues reached $5.6 billion in 2012, and are expected to continue climbing to an estimated $16 billion by 2016 (Enterprise Post). The number of mobile gamers in the United States grew 35% from 2011 to 2012, and current numbers suggest that 100 million Americans play mobile games [1]. Developers have taken notice, and the current marketplace reflects their attempts to capitalize on mobile gaming's popularity. Of the 600,000 apps available on Apple's iTunes store in April 2012, games were the most frequent type (there were 105,292, making up approximately 17% of the market) [2].

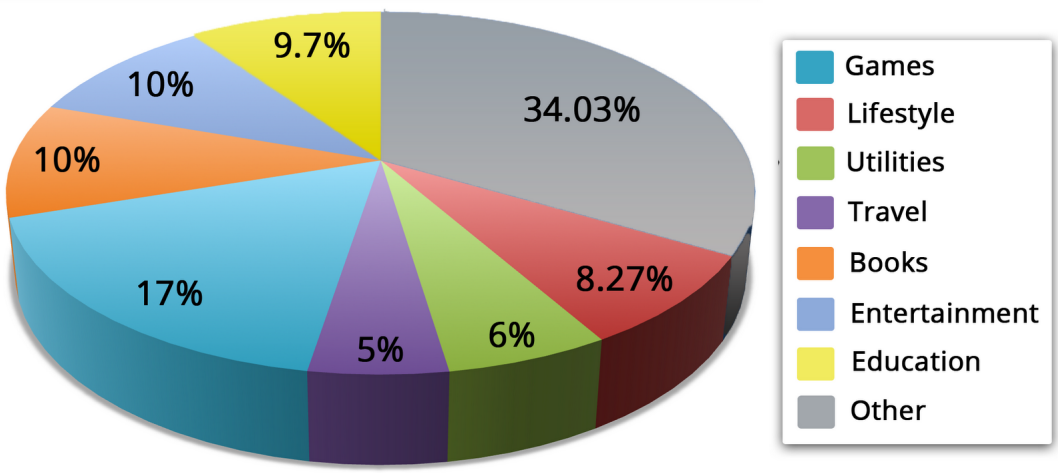

Figure 1: *Application type distribution on the Apple's App Store, May 2012.*

This upward mobility is encouraging for the gaming industry, but it also limits opportunity for small and independent game developers. These entities are much less likely to win user attention in an inundated marketplace over their more recognizable

competitors, and gaming enthusiasts are less easily able to unearth hidden gems. Consequently, there is a great need for an alternative distribution channel that can help improve discovery for gamers, and provide independent game developers with a viable platform for exposure.

Any such system must integrate a social layer in order to be successful. Social interaction is an indelible part of most modern digital products, and as many scholars have noted, it plays a critical role in gaming as well, "We see games and play increasingly embedded in social networks, in mobile phones, on websites," [3]. Social elements can enrich the gaming experience, encourage further gameplay and help to disseminate a game to a broader audience. This provides great value for both those playing the game players and developers.

For our final project, we designed and built an information system (called GameBadger) that adds social features to fully developed, single-player mobile games.

2 Economics of Information

2.1 Economies of scale

GameBadger's social features deliver powerful network effects that can help the application grow. The application allows one user to access their Facebook network and challenge people within it to a bout with a mobile game. If a challenged user does not have GameBadger installed, that individual will receive a Facebook notification with a link provided to download the application. This enables the swift propagation of the platform, since these newly minted users have an entirely different network of users that may be exposed to GameBadger.

If quantified, the growth rate for this kind of network would be close to O(n log n) [4]. It is unlikely that every friend in a given user's network will adopt the application

(this would resemble O(n2)). However, it's fair to assume that, for every user, the application will percolate down to a subset of their friends.

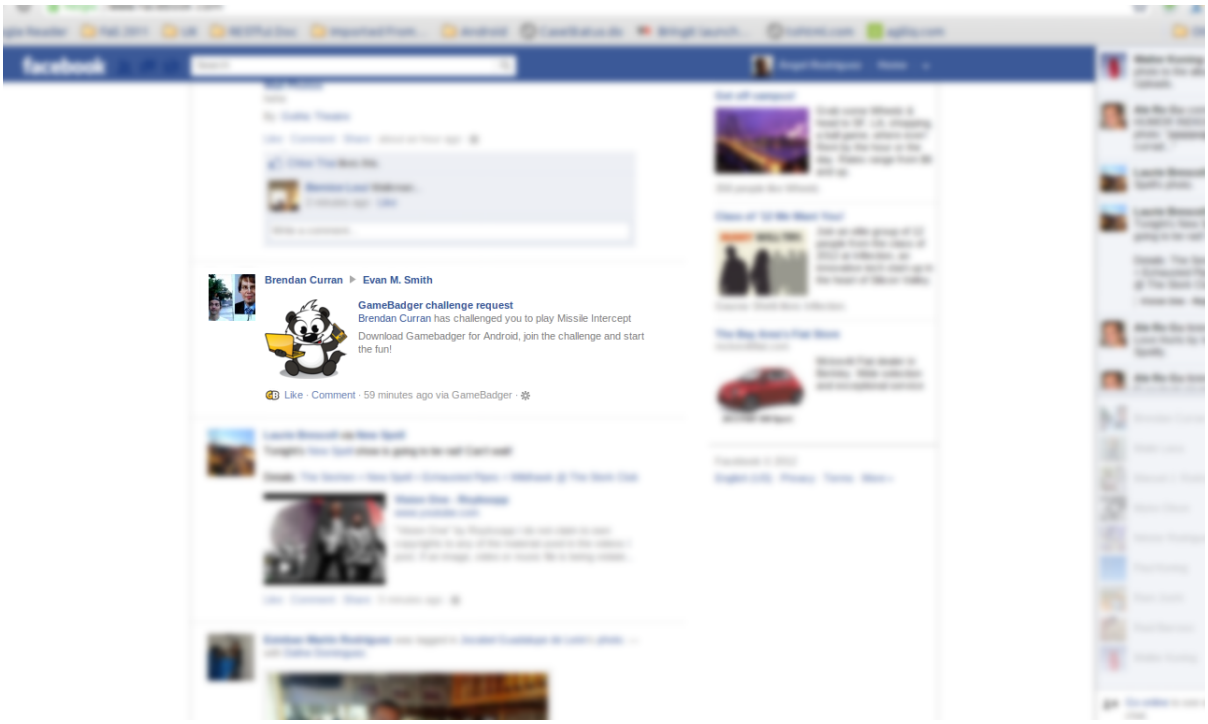

Figure 2: *Facebook post after issuing a challenge on GameBadger*

The network effect will not be inhibited by price. We would not plan on charging users to access GameBadger, but we would probably build in some kind of virtual currency conversion to help monetize the application. The same would be true for game developers, although some kind of freemium model might be adopted that would require these entities to pay a subscription rate if their user base exceeded a given benchmark.

2.2 Competition

Although there is no dominant platform for social gaming, there are several well established players that are presently competing for market supremacy. These enti-

ties include companies such as *Sometrics* (which reaches 250 million users), *OpenFeint* (115 million), *Papaya Mobile* (50 million), and *Heyzap* (4.6 million). Sometrics helps developers monetize their games through a standardized virtual currency system, and also provides in game analytics that can help developers better understand the network effects of their existing social features. OpenFeint, Papaya, and Heyzap are more similar to one another. These companies have developed APIs that expose leaderboards, badges, and other barometers of performance to users. The interaction with the tool is socially oriented, so a player may compare their own performance against their friends.

Drawing users away from these products is a daunting prospect. Their virtual currency systems create a significant switching cost, since many users will have a continual positive balance in their virtual accounts and may be unwilling to abandon their investment. Users have also paid for platform-specific games that will probably not be transferable to a different social gaming platform. Also, many users may find it more valuable to socialize within their robust existing networks, rather than risking an unknown entrant with few users.

However, these difficulties can be overcome with careful strategic planning. For example, GameBadger might offer a set amount of free currency for several months after the product's launch date. Market research could identify the most popular games in competing platforms, and a subsequent effort could be put forth to port over those games to the GameBadger network. Finally, GameBadger's challenge system is a unique feature, and does not exist in the other platforms. Since our system revolves around this feature, it may help us convert users from other platforms.

2.3 Value proposition for Game Developers

Many independent game developers are passionate about engineering games, but they are often uninterested or incapable of working on gaming's "secondary" aspects. These include marketing campaigns, social features, and metric analysis tools. All three can dramatically improve a game's success, but it takes time and expertise to maximize their effectiveness–two things that a small scale developer may not have.

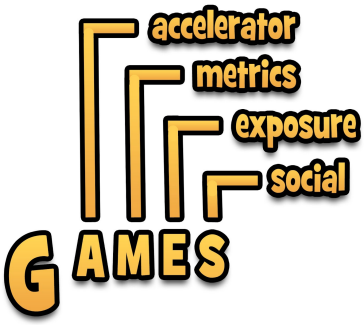

Figure 3: *GameBadger's value proposition for Gamers*

GameBadger's value proposition to game developers is simple: we allow developers to focus on the 'G', while we take care of (a)ccelerating the growth of the game through (m)etric analysis and (s)ocial functionality. This allows the developer to drive improvements for the game itself.

2.4 Value proposition for Gamers

Gamers cannot win glory in achieving a high score or beating a level if no one gets to hear about it. Gaming achievements are essentially stripped of meaning when there is no socially driven accompaniment. GameBadger provides an audience and a trophy case to validate gaming results. The application also delivers a discovery engine for future gaming conquests.

GameBadger's challenge system is its flagship feature. Challenges function somewhat like the promoter of a boxing match: they establish the terms of a contest between participants (including the game and stakes at play), track its progress, share the results with the world, and award the winnings accordingly. However, unlike boxing matches, GameBadger challenges are asynchronous. In other words, participation does not require simultaneous game playing. Currently, there are no major social gaming platforms that have integrated a challenge interaction. We believe this is the principle way GameBadger can distinguish itself from existing systems.

3 The GameBadger Application

The application's menu bar provides four different screens: **Home**, **Create**, **Games**, and **History**.

3.1 The Home Screen

Users create challenges through a standalone application that we developed for mobile devices running the Android OS. When a user launches the application, they are presented with a scrolling list of pending and active challenges on the home screen. The active challenge list provides users with records of each current challenge, and also provides a launch point for the associated game. The pending challenge list enables users to view a complete set of challenge invitations, along with the functionality to accept or reject those invitations.

3.2 The Create Screen

The create page facilitates challenge creations. There is no limit to the number of challenges a player can issue, but creating a challenge requires the following inputs:

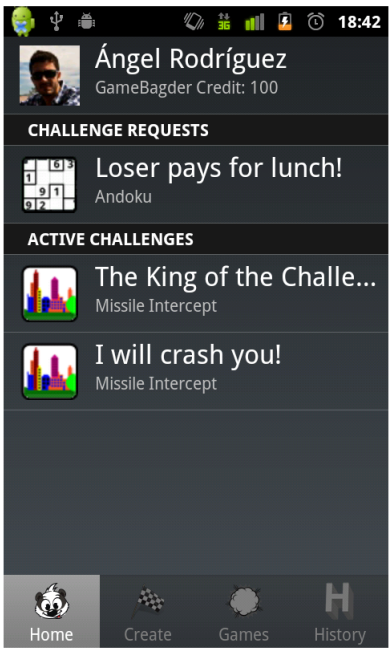

Figure 4: *The Home Screen*

1. **Challenge Game**: Users must identify a game. The choice of games is limited to those that have integrated the GameBadger API. For the purposes of a minimum viable product, we limited the choices to three open source games available for the Android OS.

2. **Challenge Title**: Challenge titles are mostly cosmetic –user input might be something like "Battle Royale" or "Loser pays for lunch¡' This value is intended to provide a layer of personalization, and also helps users distinguish among the multiple simultaneous challenges that may be active.

3. **Challenge Duration**: The user must input a date of expiration for the challenge. The GameBadger API will accept score updates until the date and time chosen occur.

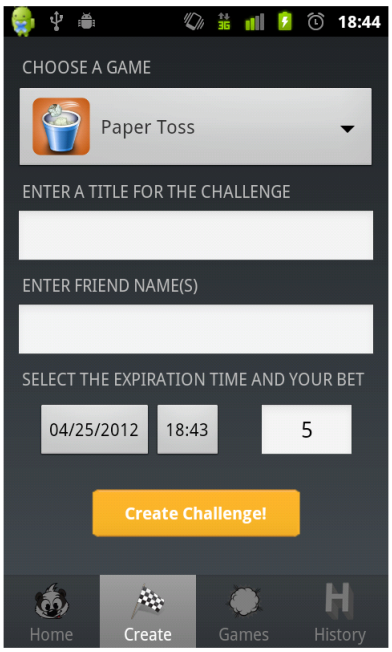

Figure 5: *The Create Screen*

4. **Challenge Participants**: When creating a challenge, users will have access to their complete list of Facebook friends, and may invite any number of them to a challenge.

5. **Challenge Bet**: A player must wager GameBadger-specific virtual currency (called "gold coins") when creating a challenge. The challenge winner collects the sum of this virtual currency once the challenge expires.

Once a challenge has been created, it becomes active. Players can access the selected game through the GameBadger application, or natively through their phone. The GameBadger API reports the player's score to the GameBadger server at the end of each gaming "session". The terms of a session will vary widely from game to game, so we allow the developer to determine when it makes the most sense to report a score.

Once the challenge expires, the API computes the highest score, adjusts each player's virtual currency, and pushes a notification.

3.3 The Games Screen

The games screen provides a digital library of GameBadger affiliated games that exist on a user's phone. A simple recommendation engine is also included to suggest new games to a user, based upon their gameplay history. By providing an additional access point, the games page encourages further gaming and helps users to distinguish the games on their phones that are capable of facilitating GameBadger challenges. It also provides game developers with an additional stage for exposure.

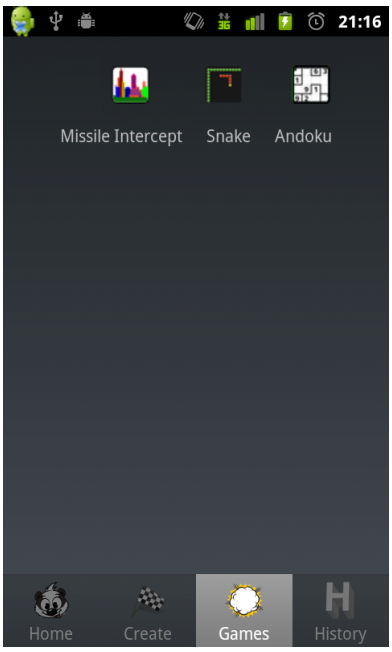

Figure 6: *The Games Screen*

3.4 The History Screen

The history tab displays a scrolling list with the results of past challenges. Users can explore the final details of their former contests, including the participants and the winning score. Each challenge record also includes a rechallenge button, which automatically regenerates a new challenge with the terms of the former one.

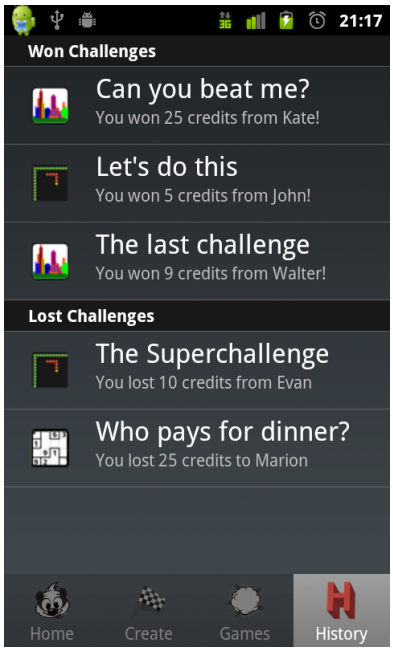

Figure 7: *The History Screen*

4 Methodology

4.1 User needs assessment

Before engaging in any engineering or design efforts, we conducted three informational interviews with people who we identified as potential GameBadger users. All three subjects regularly played mobile games and engaged with social tools on the

web, but did not use any social platforms specifically developed for mobile games. One of our subjects was a mobile game developer looking to break into the Android marketplace. We sought to identify what prevented each of our subjects from engaging with existing products in this space. Our main goal was to identify how GameBadger could eliminate those barriers to entry, and differentiate ourselves in an increasingly crowded landscape.

From our discussions, we learned that many of the existing social products for mobile gaming simply don't work very well. One user complained that the interface was not intuitive, and the amount of effort required to actually facilitate a social interaction via a competitor's application was simply not worth it. Another noted that the OpenFeint app struggled to connect with its server every time she tried to access it. This led us to two major takeaways:

- **UI Design should be simple and direct**: People who might engage in social gaming want it to be a painless and easy process. After all, their focus is on gaming. If a social layer interferes with that goal, they're likely to abandon it.
- **Engineering should be scalable**: User patience is thin when it comes to performance. If we build something that doesn't operate well, people are likely to abandon the product and never return. While users are mostly unlikely to pay close attention to the product's architecture, this will be a vital point of concern for game developers.

4.2 Personas

To meet each of these goals, we designed two primary personas for GameBadger. This effectively helped us to personify each of the concerns outlined above into two distinct targeted user groups: gamers and game developers.

Ashley Flowers

- **Age:** 28
- **Location:** Palo Alto, CA
- **Profession:** Independent Game Developer
- **Interests:** Burning Man, New York Times, IP activism

Jaime Hernandez

- **Age:** 26
- **Location:** San Francisco, CA
- **Profession:** Marketing Manager for The Gap clothing store
- **Interests:** Napa Valley, Las Vegas, restoration hardware

Table 1: *Personas used for the development of GameBadger*

The gamer persona (identified as Jaime Hernandez below) is principally concerned with the interaction of the UI. He doesn't care how the application functions–it's more interesting for him to have a social experience that's cool, and worth sharing with his friends. To meet Jaime's needs, we employed mobile usability heuristics and focused on making the interactions as seamless and simple as possible. Our product cannot be successful if people like Jaime encounter the same problems with our app that they experienced with our competitors.

Ashley Flowers became our persona for game developers. While a good UI is important to her, it's not quite as relevant as GameBagder's API and underlying network architecture. Ashley is looking for something lightweight and easy to integrate into her

game. She didn't have time to develop social features in her product, so she isn't interested in integrating something that will be difficult to learn, or create a performance burden on the game itself. It's also important for her to feel reassured that the system interacting with her game can scale quickly. This kind of avenue of exposure would quickly turn disastrous if thousands of gamers were to witness an extended period of inoperable activity due to the failure of someone else's server.

4.3 Prototyping

We developed our prototype through a series of brainstorming sessions and whiteboard sketches. At http://www.gamebadger.com/usertesting, an online prototype optimized for mobile devices can be found. Once the idea started to become more concrete, we shifted to Balsamiq in order to consecrate our ideas digitally. Our UI design focused on eliciting challenge creation and curation from users while minimizing the amount of effort required to complete such a task. We also adhered to general mobile UI design principles, including the following:

- **Provide relevant options for exploration and pivoting**: Some users will access the application in order to do a 'deep dive' of the information contained therein. That might mean accessing past challenges to track their overall gaming trajectory, or it might mean perusing the games available through the GameBadger platform. However, just as often it's likely that users will want to do something easily and quickly: that might mean responding to a challenge invitation, or participating in one. Keeping this in mind, we designed our UI to facilitate both options in an easy manner.

- **Emphasize content over navigation**: With a limited screen real estate, it is important to maximize every pixel in order to deliver content. Navigational tools

are important, but options should be minimal and, as much as possible, should avoid interfering with the forefront of the application's screen.

- **Align with how people use their mobile devices and why**: Mobile usage implies a highly distracted user state. We need to assume from the outset that we are competing for the user's attention against a bevy of stimuli outside the UI itself, since most users will likely access the application while commuting or waiting for something different. As a consequence, it was imperative for us to design an interface that minimizes complexity and superfluity.

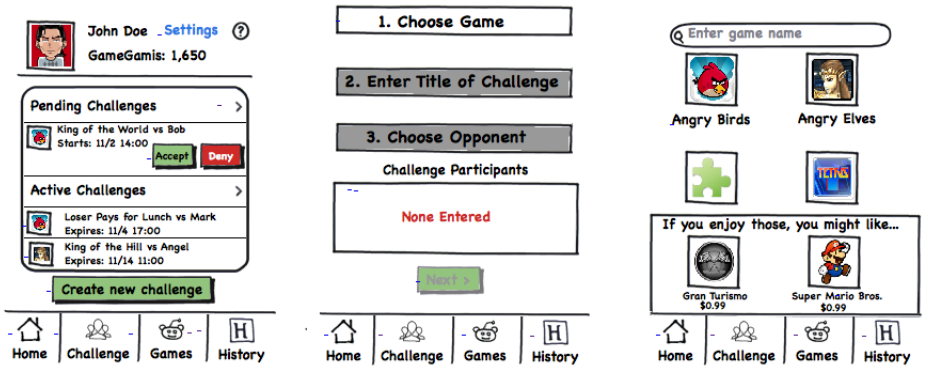

Figure 8: *Prototype screens designed during the early stages of the product.*

Taking an iterative design approach helped us to strip unnecessary features from our eventual Minimum Viable Product. For instance, we removed an in-app store that would have facilitated purchases of virtual currency. We also redacted plans to allow different kinds of challenges aside from highest score.

4.4 User Testing

We spoke with four individuals during the early part of the semester. Two of these four were avid gamers without much technical expertise, and closely fit our Jaime

Hernandez persona. The other two in our group were game developers, who were real life manifestations of Ashley Flowers. After adapting our prototype for mobile devices, we presented it to the "Jaime's" and asked them to step through the challenge creatio process on their smart phone. As they explored the application, we observed their responses, took note of physical cues, and interviewed them about the experience afterwards.

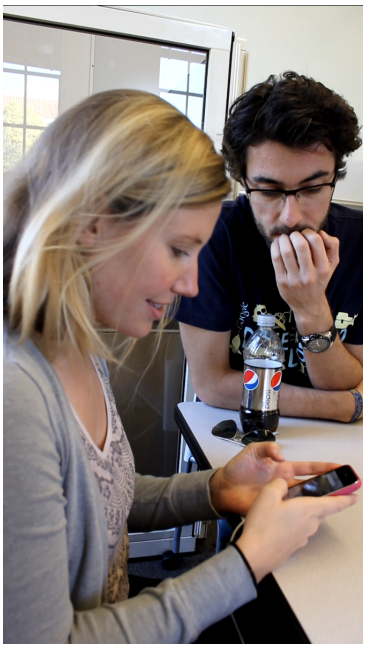

Figure 9: *User testing interviews.*

User feedback for the protoype UI was mostly positive. People responded well to the history tab, and the hierarchy of pending and active challenges made intuitive sense to our subjects. However, our users did point out a few problems with the UI choices we laid out in the prototype:

- **The two step process for creating challenges was unpopular**: We received feedback indicating that a two part user flow made the challenge creation feel protracted and less engaging. We therefore consolidated both screens into one long scrolling screen.

- **Our virtual currency confused test subjects**: We learned that the points within a game itself and our system-specific virtual currency (also called 'points' at the time) were being inadvertently conflated. To help ease confusion, we changed the name of our in-game currency from 'points' to 'Gold Coins'.

For the second phase of user testing, we met with two mobile game developers who fit the "Ashley Flowers" persona. Although we showed these subjects the prototype, our conversations were more focused on the value proposition that our system might present to someone in their position. We emphasized the elements of our system that might contribute towards their success, and asked for their input on the existing system's mechanics. At the end, we asked them to answer the following questions:

- Did they think our API looked like it would be easy to integrate?
- How did they feel about our standalone app? Did they feel like it might compete with their own app for the user's attention?
- Were the benefits of GameBadger integration obvious and alluring?
- Would they be interested in a potential partnership at some point in the future?

Both developers expressed enthusiasm for the concept we presented, and were very interested in partnering with us once our product reached a viable state.

5 Information Architecture

GameBadger's architecture is divided into two components: a native Android application, and the server, which listens for API requests and manages the database appropriately.

The figure above illustrates the system architecture. The numbers represent the sequence in which actions take place:

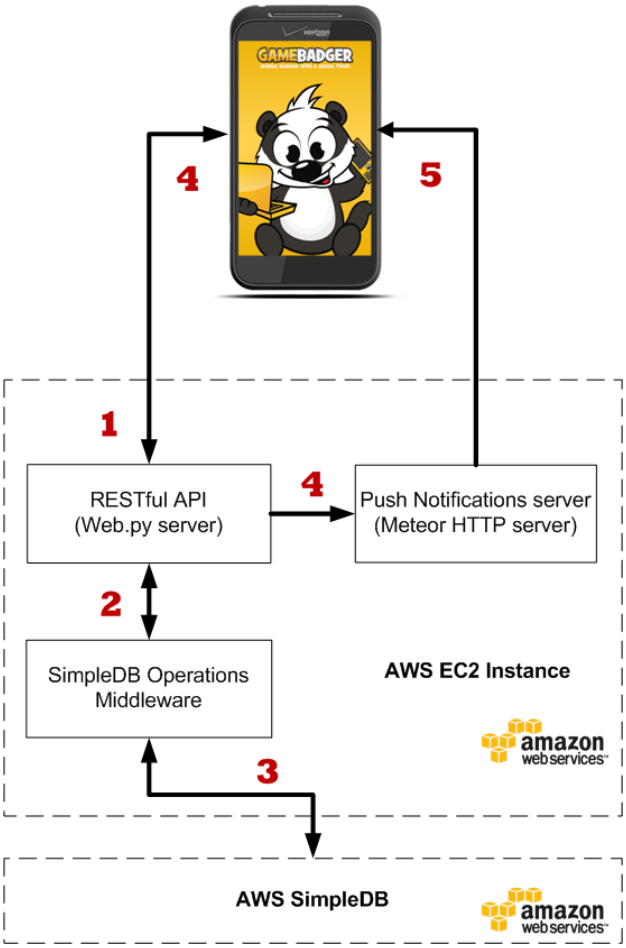

Figure 10: *System Architecture*

1. The Android application calls any of the methods of the API. On the server side, the RESTful API web server is listening on a port on the AWS EC2 instance and process the request, that can be a `GET` or a `POST`. If it is a `POST`, it will also have a JSON object associated with the request.

2. If the API needs data from the database, it will call any of the methods in `sdbops.py`. That way we separate the web server from the structure of the database in different modules.

3. The actual query to the database takes place.

4. The web server evaluates if as a consequence of the call, a push notification should be sent to the device. In that case, another web server –the one handling the notifications– is triggered.

5. Data is returned to the client in the form of a JSON object.

5.1 The GameBadger Android Application

GameBadger's Android application was written in Java, and developed through the Eclipse IDE[1]. Since the Android SDK has no native push notification system, we integrated the Deacon[2] push notification library to facilitate notifications. Deacon uses the Meteor web server to flexibly deliver notifications using comet technology.

Meteor[3] is an Open Source HTTP server written in Perl that facilitates integration with streaming data feeds. The library listens on one port for event controllers, and on another for channel subscribers. Event controllers are clients that employ Meteor's command protocol to inject events into named channels. Each Android device will be assigned a unique channel that the server employs to push notifications to the client.

5.2 The GameBadger API

GameBadger's API handles the transfer of application data between the server and Android devices. This is especially important for the games involved and their challenges. The games continuously report scoring results to the server, so the challenge can be managed appropriately. However, this interaction adds substantial overhead to the server, and could easily overflow a system without the proper design to handle such a load.

[1] http://www.eclipse.org/
[2] http://deacon.daverea.com/
[3] http://meteorserver.org/

GameBadger's API solves this problem by listening to messages sent from other processes. This action runs on the background of the device itself and reports data from the games, including the game runtime, the achievements unlocked, and the recorded score. Since the API was written in a RESTful architectural style, there is no need for a client to login. REST employs HTTP's stateless nature, so the system does not keep track of sessions.

The API's server runs on an open source web framework called Web.py built in Python. We chose this particular system based upon its lightweight characteristics. Web.py is very well respected in the open source community, and runs on well known websites such as Reddit.

The Web.py[4] framework serves a Python file named `gbapi.py` that contains forty one methods. Each method belongs to a custom written `GET` or `POST` class, and allows the Android client side application to read and write data on the server.`GET` requests take parameters from the same URL, but `POST` requests require parameters within the URL and in JSON. For example, the following would be a valid `GET` Request:

http://ec2machine.amazonaws.com:1234/getActiveChallenges?fbid=998877

Whereas the POST request URL below requires JSON formed as follows:

http://ec2machine.amazonaws.com:1234/postChallenge

and should have the following JSON attached in the request:

[4]http://webpy.org/

```
{
    "gameId":"10912",
    "challengeName":"King of the World",
    "endDate":"11/2/2012 14:00",
    "goldCoins":"3500",
    "opponents":[
        {
            "userID":"1234"
        },
        {
            "userID":"5678"
        }
    ]
}
```

The API defines a child class for every method, which inherits the main attributes and methods of its parent. The following is an excerpt of the abstract request class:

```
class AbstractRequest(object):
  #Abstract database that handles the
  #high-level HTTP primitives
  def GET(self):
      return self.get_resource()

  def POST(self):
      data = web.data()
      print data
      self.put_key(data)
      return True

  def DELETE(self, name):
      self.delete_key(str(name))

  def PUT(self, name=None):
      #Creates a new document with the request's data and
      #generates a unique key for that document.
      key = str(uuid.uuid4())
      self.POST(key)
      return key

  def get_resource(self):
      result = self.get_values()
```

```
      web.header('Content-Type', 'application/json')
      if result is not None:
          return result
```

The PendingChallenge class, for example, handles two requests: one `GET` and one `POST`. This is a good example of how inheritance works in our API.

```
class PendingChallenges(AbstractRequest):
  #This handles GET requests
  def get_values(self):
    request_params = web.input()
    challenges = []
    challengesList = sdbops.getPendingChallenges(request_params.fbid)
    for i in range (0, len(challengesList)):
       challenges.append(sdbops.getChallenge(challengesList[i]))
    return json.dumps(challenges)

  #This handles POST requests
  def put_key(self, data):
    # For a POST request, parameters are passed via JSON
    request_params = json.loads(data)
    challengeID = sdbops.createChallenge(request_params["fbID"], \
       request_params["gameID"], request_params["bet"], \
       request_params["expirationDate"], \
       request_params["challengeName"])
    opponents = request_params["opponents"]

    # We iterate the list and extract the userIDs,
    # which are added to the challenge
    for item in opponents:
       sdbops.addUsersToChallenge(item["fbID"], item["username"], \
          str(challengeID))
    return True
```

The GameBadger architecture also includes an intermediate layer between the API calls and the datastore. Methods that handle the exchange between the two are contained in a file named `sdbops.py`. Decoupling the API from access to the database allowed us to isolate possible errors and improve troubleshooting. Since the server does not maintain sessions, each database connection is autonomous. This creates

some performance problems, since access time is increased, but we plan to mitigate this by limiting the number of accesses and methods present within this intermediate layer in the future. The file includes twenty four methods and employs a python module called `Boto` to facilitate interaction with the database. Here is a sample code of how to use Boto to retrieve data from the database:

```
def getGameName(gameID):
  sdb = boto.connect_sdb(_access_key_, _secret_key_)
  dom = sdb.get_domain('Game')
  query = 'select * from `Game` where GameID="' + str(gameID) + '"'
  check = dom.select(query)
  return str(check.next()['GameName'])
```

Once the due time of a challenge is reached, its winner is computed and notified. To do this, we schedule a task every time a new challenge is created. It is similar to a cron job, where the function `job` is executed at the specified time – the challenge due time:

```
sched.add_cron_job(job, month='*', \
   day='*', hour='15', minute='29')
```

`job` was previously defined in the code as follows. It gets the winner FBID of the challenge and calls the script that will push a notification to the user device, letting him now that he won the challenge.

```
from apscheduler.scheduler import Scheduler

# Start the scheduler
sched = Scheduler()
sched.start()

def job():
   winner = sdbops.getWinner(challengeID)
   os.system("python pushnotification.py " + winner + '"')
```

5.3 The GameBadger Server

For storage and computing power, we used Amazon Web Services (AWS). The GameBadger API runs on an EC2 server, which provides instant access to a Linux virtual machine.

We chose to use AWS for the following reasons:

1. **Low cost**: AWS follows a pay-as-you-go pricing model, which includes a free tier for low-traffic server space.

2. **Scalability**: AWS has the capacity to scale quickly, as user bases and server requests increase.

3. **Security**: The safety of the data stored at Amazon services complies with internationally recognized certifications and audits. By leveraging this, we can focus on building out product features.

5.4 The GameBadger Database

AWS also provides a database model named SimpleDB, which we chose to use as our storage method. SimpleDB is a nonrelational database that offloads the work of database administration. This approach was advantageous during the early stages of the project, since it is flexible and does not require any remodeling as our modeling and requirements changed. Additionally, the volumes of requests that a NoSQL database can handle are significantly higher than the ones in a traditional one, due to the transaction rates and a low-cost commodity hardware design. However, NoSQL does have some drawbacks. For example, joins –a powerful tool in relational databases– are not supported. This means that data must be denormalized and precalculated.

6 Conclusion

GameBadger was designed to be a minimum viable product, and as a result, there are many considerations that we were forced to table for future development cycles. Security, for instance, is an extremely complex consideration with this kind of application. While it is also of paramount importance, we simply did not have the capacity to engineer against malicious behavior (beyond the power of AWS's cloud-based storage). There are also several social features that we discussed in brainstorming sessions, but simply could not build into the product given our time constraints. For example, user leaderboards and tournaments would provide additional social incentives, and encourage further game play from GameBadger users. However, such a system demands careful consideration and meticulous engineering, and are better suited for later versions of the application.

Also, because our focus lay on the actual architecture for our virtual currency, we did not have much time to consider how users might be able to apply it. There are quite a few ways to make this exciting: we could translate our virtual currency into our partner's game titles; we could allow users to redeem GameGami virtual currency for Facebook credits; or, we could establish partnerships with physical businesses, and allow users to exchange their virtual currency for real world goods. There is also much potential in the way of data analytics: quantifying user engagement, tracking transactions, and monitoring subscriptions may, in the future, allow us to further increase the value proposition delivered to game developers and make our information system even more valuable.

References

[1] Newzoo Trend Report, *Mobile Games* , 2012

[2] 148apps.biz, *iOS development news and information for the community, by the community.*, April 2012, http://www.148apps.biz

[3] Celia Pearce, *Communities of Play: Emergent Cultures in Multiplayer Games and Virtual Worlds*, Sep 2009, MIT Press

[4] Bob Briscoe, Andrew Odlyzko, and Benjamin Tilly, *Metcalfe's Law is Wrong.* July 2006 IEEE Spectrum. Points out that Metcalfe's Law is wrong, that the value is closer to n log (n)

[5] Enterprise Post, *Propelled by in-app purchases, mobile gaming revenue to hit $16 billion in 2016*, August 2012, http://enterprisepost.com/

[6] Law, Edith and Luis von Ahn, *Input-Agreement: A New Mechanism for Collecting Data Using Human Computation Games*, CHI 2009, 1197-1206

[7] Richardson, Ingrid *Ludic Mobilities: The Corporealities of Mobile Gaming*, November 2010, Mobilities Vol. 5 Issue 4, p431-447.

[8] Soh, Jason O.B. and Bernard C.Y. Tan., *Mobile Gaming*, March 2008, Communications of the ACM; Mar2008, Vol. 51 Issue 3, p35-39.