Playfish's Social Gaming Architecture - 50 Million Monthly Users and Growing

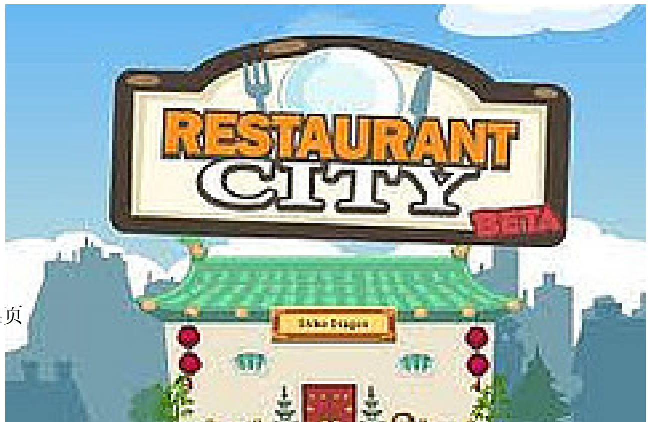
Tuesday, September 21, 2010 at 10:30AM

Todd Hoff in Example

Ten million players a day and over fifty million players a month interact socially with friends using Playfish games on social platforms like The Facebook, MySpace, and the iPhone. Playfish was an early innovator in the fastest growing segment of the game industry: social gaming, which is the love child between casual gaming and social networking. Playfish was also an early adopter of the Amazon cloud, running their system entirely on 100s of cloud servers. Playfish finds itself at the nexus of some hot trends (which may by why EA bought them for $300 million and they think a $1 billion game is possible): building games on social networks, build applications in the cloud, mobile gaming, leveraging data driven design to continuously evolve and improve systems, agile development and deployment, and selling virtual good as a business model.

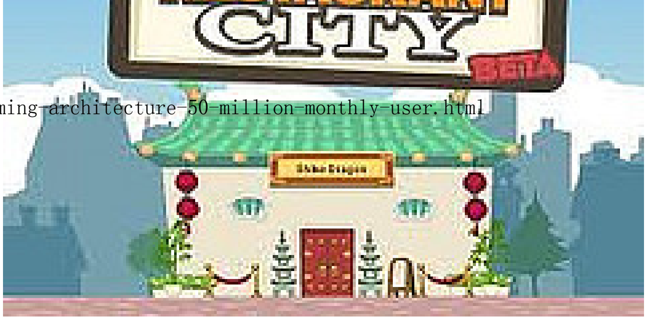
How can a small company make all this happen? To explain the magic I interviewed Playfish's Jodi Moran, Senior Director of Engineering, and Martin Frost, Chief Architect, first Engineer and Operations guy at Playfish. Lots of good stuff, so let's move on to the nitty gritty.

Site: playfish.com



Stats

1. 50 Million Monthly Active



Users
2. 10 Million Daily Active Users
3. 100s of Server Machines
4. 10 Games, more being released all the time
5. 200 million games have been downloaded, installed, and played
6. 100:1 ratio of servers to operations people
7. About 200 employees and 250 contractors running out of 4 studios

Platform

1. Flash on the Client Side
2. Java on the Server Side
3. Some PHP for tools
4. Amazon: EC2, CloudFront, S3, Hadoop/Elastic Map Reduce, Hive, some SQS, some SimpleDB
5. HAProxy for load balancing
6. MySQL for sharded, blob storage
7. Jetty Application Servers
8. YAMI4 for service transport. Gives point-to-point connectivity, low latency.

Key Forces

What are the key forces that shape Playfish's architecture?

1. **Games are free-to-play (F2P)**. Playfish games are free-to-play: they don't cost money to play, but they still cost money to run. The consequence of this model is that it's not possible to simply throw hardware at problems. Costs must be controlled, so they try to run

lean and efficient. With other game models monthly subscriptions subsidize hardware and product investment. MMOs (massively multiplayer online game), for example, run hundreds or thousands of users per server. Playfish targets orders of magnitude above that. MMOs provide a server intensive game experience, but the only way they can to do that is by charging $10-$50 a month subscription fees, which is a completely different space.

2. **Games are social**. Playfish focuses on social games, which means it's about you engaging and interacting with your friends. In fact, you can only play with your real friends, your social graph. Games are not really played to win. Results are ordered by points and level, but people play more to express themselves and show off their design skills. It's a different type of gaming. In Restaurant City, for example, the goal is not to be the richest or the best, but to have the most creative and expressive restaurant. Newer games don't even use global leader boards anymore. Older games used a global leader board where everyone competed against everyone else. Games were structured more traditionally with challenges and a "look my score it's bigger than yours" attitude. Now they have a leader board, but it's just between you and your friends. It's more intimate now. People don't care about if they are better than someone in a different country, they care if they are better than their friends.
3. **Games are asynchronous**. Asynchronous games are games where players can play at different times. Since your friends are rarely all on-line at the same time, social games are played asynchronously. You don't all have to play at the same time. In a MMO like Worlds of Warcraft, for example, players exist in the same space at the same time and they can act simultaneously, which makes them very latency sensitive and difficult to scale. Playfish's games are different, they are single or multiplayer games, played asynchronously. Players don't exist in the same game space at the same time, they play in their local client. While you play with your

friends, you are not blocked on your friends, you can make progress on your own. This model leads to all kinds of specialized design possibilities. Examples of asynchronous game play are familiar games like Scrabble or Texas Hold'em, where players take turns. More innovative are games like Playfish's Biggest Brain, a game where friends keep trying to prove who is smarter, uses a leader board to display pictures of your friends and their scores, so you can always see who is in the lead. It's asynchronous because each player plays independently, yet the leader board allows all the players to check each other's progress.

4. **Games are casual**. Asynchronous games can be hardcore strategy games, something like the Diplomacy board game from the 1950s. Social games tend to be casual games that are really easy to play, can be played with a few mouse clicks, in short bursts of time, from any location. Playfish extends this model with high production value 3D graphics, easy controls, customized characters, and multiplayer challenges, which leads to very high user engagement.
5. **Scale is limited by the size of individual social networks**. Contrary to other application areas where social means hard to scale, for Playfish social isn't just a scaling problem, it's also a scaling solution. Playfish games are not global in the sense that a million people don't try to play the same game at the same time. Playfish games are social. They are played with your friends more for pleasurable interaction than down and dirty competition. The consequence of this is that individual games are inherently scalable because they involve so few players. Take, for example, a global leader board with 50 million users. That's a lot of writes on a single data structure and that may require sharding to make it scale. A leader board for a social network of a few players is much more straightforward to implement.
6. **Large numbers of users**. The shear number of users actively

playing games puts pressure on their architecture to scale as a total system. Each individual game may be relatively easy to scale, but with so many active games, the complete system is not so easy to scale.

7. **Rapid growth**. By design these games are viral because they exploit the social graph. They are also fun, easy to play, and help maintain relationships between friends. All these factors contribute to social gaming's rapid expansion, which means Playfish don't have to just deal with a lot of users, they have to deal with a continual stream of new users.
8. **Rapid change**. The game market moves rapidly and there's lots of competition. This also is a very new field, everyone is still learning what works and doesn't work, there's a lot of churn. Playfish can't wait on long procurement processes, long design cycles, or long release cycle. They must stay agile.
9. **Hit driven nature of games**. It's difficult to predict which games will be successful, so when a game does take off it must be able to expand immediately. There's no time to bring on new resources when a game hits it big. To allocate resources for the maximum projected growth for each of their games simply wouldn't work. They have too many games, which would require an absurd amount of servers that would most likely go unused.
10. **Experimental game release**. Games can be seen as experiments targeted at a niche. Some experiments will be more successful than others. This gets back to the very agile nature of the social game industry.
11. **Multiple games**. Playfish is not a single product company. Other companies support a single game. Playfish must support and develop multiple games simultaneously, with new games always pushing on the schedule. This puts a lot of pressure on development and operations.
12. **Smart client**. Playfish games are written in Flash, which is a very

capable rich client. It can intelligently cache data, support local operation, and remove a lot of load of the servers.

13. **Games are applications, not web sites**. Since the smart client removes many of the reads, most of the database activity is **write heavy**. 60% of the load on a typical database master is writes, whereas most web sites tend to be read heavy. Since games are played over the web it's easy to think of them as web sites, but they are really applications delivered over the web.
14. **Heavy load time latency**. Playfish games are global in sense that they have users all over the world. Because the game is played asynchronously in a smart client, game latency is not critical, which means when a game is loading--flash game + assets--is the most noticeable latency the users experiences and they must reduce this as much as possible.
15. **Real-time feedback from users**. Social games are digitally distributed over social networks via the web, completely bypassing the traditional distribution channels through stores or telcos. For the first time there's no middleman controlling game distribution. Any user can just sit back at home and play when and with whom they want. Game makers can now connect with their audience in a way that was never possible before. It's possible to connect directly with a user base to help evolve and make games better. Users can give game designers real-time feedback and effect the design of games they play.
16. **Social games are a service, not a box**. Traditional games are bought in a box, on a shelf, and distributors control the channel. They sell into the same 25 year old hard core gamer demographic. Social games are expanding out into new markets, attracting people who have never played games before. Games that are a service create a very long relationship between players and the creative team behind the game. It's possible to do weekly releases, nurture the game along, and continually learn what players want.

This wasn't possible before when you had multi-year development cycles.

17. **People are disconnected and are seeking ways to connect**. Pet Society is a virtual world where players decorate their house and their friend's house. Around Christmas time they started selling virtual Christmas and ornaments for $2 each. They sold $40 million dollars worth of virtual currency. Big believers in behavioral analysis, they asked their users why they would do that? It came down to connecting with people they were disconnected from. Previously they would have a tree in their home and people would see the tree and their creativity. Now people are all disconnected so 3-4 friends would see the real Christmas tree whereas all their friends would see the virtual Christmas tree on Facebook. There's a bigger bang for your virtual buck in terms of reaching people. What drives purchases is the desire to socially express themselves, virtual goods are the means, and social networks are the new territory. Virtual gifts are bought for the perceived value in the interaction with your friends, just like in the real world.
18. **Background simulation**. Games, like Restaurant City, are starting to feature a background simulation component that runs even when the user is not online. This causes the game to evolve in such a way that encourages player to check back regularly. In Restaurant City, for example, players need to make sure restaurant employees are being fed. This simulation component is an interesting new load that must be integrated into the game infrastructure.
19. **Need to scale in several different dimensions at once**. Playfish needs to scale to support more users, more games, more data per user, more accesses per user, more development staff, and more operations staff. The hardest thing to scale is scaling the people that support everything.

Key Scaling Strategies

In response to these forces, Playfish has followed a few key scaling strategies:

1. **Fun**. To drive growth the key design metric is to create a game so fun that it generates such strong emotions that people feel they simply must play and must invite their friends to join to have even more fun. Emotions are maximized when shared within a social network of friends. Traditional games are immersive experiences with great sound and graphics. Playfish tries to design games for social interaction, where you get a benefit to the game by inviting friends. You could play Restaurant City on your own, but it's more fun to invite your friends to a be a cook in your restaurant. You get incrementally more fun by adding friends into a game. The desire for users to involve friends in order to have more fun is what drives greater distribution and growth.
2. **Micro-transaction based revenue model**. Micro-transactions are typically high volume, low-value transactions. To pay for the service Playfish makes money by players purchasing in-game virtual items and services. Playfish also makes money from ads and product placements. This strategy supports the free-to-play model while providing revenue based on natural game play.
3. **Cloud**. Playfish has been 100% cloud based from the very start, launching their first game as a beta on EC2 in 2007. The cloud is almost the perfect answer to a large number of forces we covered in the previous section. I won't bore you by talking about all the wonders of the cloud, but it's easy to see how elasticity helps solve many of their variable demand problems. If a game becomes popular they can spin up more resources to handle the load. No procurement processes necessary. And if demand falls they can simply give the instances back. The API/IaaS (Infrastructure as a Service) nature of the cloud easily satisfies their desire for agility,

the need to keep teams small, and requirement to release early and often. You may think with pay-to-play model the cloud is too expensive, we'll talk more about this in the cloud section, but they don't see it that way. They are far more concerned about the opportunity cost of not being able to develop new games and improve existing games in a rapidly evolving market.

4. **SOA (Service Oriented Architecture)**. They are very big on SOA. It's the organizing principle of their architecture and how they structure their teams. Each game is considered a separate service and they are released independently of each other. Internally software organized into components that offer an API and these components are separately managed and scalable.
5. **Shard**. Playfish is a write heavy service. To deal with writes Playfish went to a sharded architecture because it's the only real way to scale writes.
6. **BLOB**. Multiple records are not stored for a user. Instead, the user data is stored in a BLOB (Binary Large Object) in MySQL, so it can be quickly accessed via a key-value approach.
7. **Asynchronicity**. Writing on the server is asynchronous from game play. The user does not have to wait for writes to complete on the server to continue playing the game. They try to hide latency from the user as much as possible. In a MMO each move has to be communicated to all the users and latency is key. Not so with Playfish games.
8. **Smart Client**. By using a smart Flash client Playfish is able to take advantage of the higher processing power on client machines. As they add more users they also add more compute capacity. They have to get right balance of what is on the server and client side. The appropriate mix varies by game. The smart client caches to saves on reads, but it also allows the game to be played independently on the client, without talking to the servers.
9. **Data driven game improvement**. Playfish collects an enormous

amount of data on game play that they use to continually improve existing games and help decide what games to invent next. They are using Amazon's Elastic Map Reduce as their analytics platform.

10. **Agility**. A common thread through all of Playfish's thinking is the relentless need for agility, to be able to respond quickly, easily, and efficiently to every situation. Agility is revealed in their choice of the cloud, organizing around services, fast release cycles, keeping teams small and empowered, and continually improving game design through data mining and customer feedback.

Basic Game Architecture

1. Games run in Flash clients.
2. The clients send requests, in the form of a service level API, to HAProxy servers which load balance requests across Jetty application servers that run in the Amazon cloud.
3. All back-end applications are written in Java and are organized using a services oriented architecture.
4. The Jetty servers are all stateless, which simplifies deployments, upgrades, and improves availability.
5. MySQL is used as the data tier. Data is sharded across MySQL servers and stored in a BLOB format.
6. Playfish was an early adopter of Amazon's cloud so they were unable to make use of later Amazon developments like load balancing. If they had to start from scratch they would probably go that direction.
7. Changes are pushed to the server asynchronously. Rather than a user clicking a button and that action sent to the server to see what happened, the system is designed so the client is empowered to decide what the action is and the server checks if the action is valid. Processing is on the client side. If there is a high latency between the user and the service, then the user won't see it because of

asynchronous saving and the smart client. At the end of the day it's what the user sees that matters. The important thing is that when there are glitches in the network or higher latency, that the user has a fun game experience.

8. Playfish's first few games were simple single player games, like Who Has the Biggest Brain, with features like high score and challenges. Not very complicated and not very heavy on the server side. They only had 5 weeks from start to finish on the project. That included the learning and coding to Facebook APIs, learning and coding to AWS, coding game servers, and setting up production infrastructure. The first three games continued that pattern: high scores and challenges. That gave them some breathing room to start building out their infrastructure. The first game that changed things was Pet Society in 2008. It was the first game that had significant use of virtual items. Data storage went from storing a few attributes per user, like avatar customization and high score, to storing potentially thousands of items per user for all the virtual items. This put a lot of strain on the system. There were some big service problems in the early days as they grew very very fast. Then they put in sharding and the system became much more stable. But first they tried various other techniques. They tried adding more read replicas, like 12 at one point, but that didn't really work. They tried to patch the system as much as best they could. Then they bit the bullet and put in the sharding. It took 2 weeks from start to roll out. All of performance problems went away on that game. Over time users acquired lots and lots of virtual items, so the volume of rows exploded. Each item was stored in its own row. Even if you split users into shards they found for older users the shards would keep growing and growing, even if the number of users stayed the same. This is one of the original drivers to going to BLOBs. BLOBs got rid of the multiple rows that caused such performance problems. Over time games started getting more complicated. Pet Society

didn't have any simulation elements. Then they launched Restaurant City at the end of 2008 which had the first offline simulation element. Your restaurant continued to run and earn money while players are away. This introduced challenges, but adding the extra processing in the cloud was relatively straight forward. The simulation logic was implemented in the client and the server would do things like check for fraud.

Service Oriented Architecture

1. Playfish are really big advocates of SOA. A SOA encapsulates data and function together into components that can be deployed independently through a distributed system. The distributed components talk through an API called *service contracts*.
2. Services make sure the dependency between all the parts of the system are well known and as loosely coupled as possible.
3. Supports complexity management. The system can be composed into separate understandable components.
4. Components are deployed and upgraded independently, which gives flexibility and agility and makes it easier to scale development and operations teams.
5. Independent services can be optimized independently of other services.
6. When a service fails it's easier to degrade gracefully.
7. Each game is considered to be a service. The UI and the backend are a package. They don't do separate releases of the UI and the backend.

The Cloud

1. Playfish has been 100% cloud based from the very start, launching their first game on a beta version of EC2 in 2007.

2. Cloud allows Playfish to innovate and try new features and new game with very low friction, which is key in a fast moving market. Moving from their many read replica system to a sharding system took 2 weeks, which couldn't have been done without the flexibility of the cloud.
3. The cloud allows them to concentrate on what makes them special, not building and managing servers. Because of the cloud operations doesn't have to focus on machine maintenance, they can focus on higher value service, like developing automation across all their different servers and games.
4. Capacity is now seen as a commodity when designing applications.
5. The ratio of servers to operations people is 100:1. Such a high ratio is possible because of the cloud infrastructure.
6. Servers fail so this must be planned on from the start.
7. It's not possible to keep adding memory to servers so you may have to scale out earlier than you would like.
8. Key feature of the cloud is *flexibility*. You can be as agile as you want. You don't need to be surprised when suddenly get a lot of traffic. You don't have to wait for procurement of servers.
9. You never know how quickly a game will take off. Sometimes you do know, sports games go quickly, but other games may suddenly explodes. In the cloud that doesn't have to be a problem.
10. From the beginning there was never an expectation that they could scale-up in the cloud. Everything is designed to scale by adding more machines.
11. They can't use all the Amazon services because they had to roll their own. Switching away from their own systems that they understand would be unnecessarily risky. ELB and RDS weren't available, for example, so they had to build their own. Switching to those services now wouldn't make sense.
12. Playfish is cloud to the core. They take advantage of everything they can in the cloud. More capacity is acquired with ease. They have no

internal servers at all. All development machines are in the cloud. The cloud makes it trivial to launch new environments. To test sharding, for example, is easy, simply copy everything over with a new configuration. This is much harder when running in a datacenter.

13. The **cloud is not more costly** than bare metal when you consider everything.
    1. Bare metal may look cheaper based on bandwidth and a unit of rack space, but if you look at all the stuff you get, it would be a lot of work. Take the advanced availability features, for example. Change an API call and you get double datacenters. You can't do that in a bare metal situation. Just consider the staffing costs to setup and maintain. The cloud looks really expensive, but when you get really big capacity breaks start kicking in.
    2. The major costs to consider are **opportunity costs**. This is the single biggest advantage. For example, when they first implemented sharding in Pet Society it 2 week from start to deploy. Users were immediately happy. The speed of their implementation relied on being able fire up a whole load of servers in production and test and migrate data. If you had a two month lead time you would have had a lot of unhappy users for two months.
14. Playfish runs in multiple availability zones within the same region. Servers are relatively close which reduces latency. They aren't spread out like in MMO systems. Latency is dealt with at a higher level using asynchronous writes, caching in the client, and caching in a CDN. It can take 3 seconds to perform a game action back on the server, but because it's async, users don't notice. The CDN helps reduce what they do notice, which is asset and game loading.
15. CloudFront is used to reduce load latency. Playfish is global in sense that have users all over the world. The loading time of the

game, which includes the flash code plus game assets, is their most noticeable latency. CloudFront reduces this latency as it spreads the content out.

Database System

1. MySQL is used as a sharded key-value database to store BLOBs.
2. Users are sharded across multiple database clusters, each with their own master and read replica.
    1. There's little benefit for them to have more replicas because they are write heavy. Nearly all the traffic is writes. Writes are harder to scale. Can't cache and more read replicas don't help.
    2. In an earlier architecture they had one master with 12 read slaves, which didn't perform well.
    3. With sharding they went from one master and 12 read replicas to two masters and two read replicas, which helped with both reads and writes.
3. Sharding meant indexes got smaller which means they could fit in memory. Keeping the index in cache ensures a user lookup doesn't have to hit the disk, lookups can be served from RAM.
4. By sharding they can control how many users are in each shard, so they can be sure they won't blow their in-memory index cache and start hitting the disk.
5. They have tried to optimize the size of a user record so that more users will fit in memory. This is why they went to storing BLOBs instead of using data normalized into rows. Now there is one database record per user.
6. Work is taken out of the database server and moved to the application servers, which are very easily scaled horizontally in the cloud.
7. Most websites use scaling techniques, like memcache, for read caching are that aren't that useful to Playfish. With a Flash client

most of what would be cached in memcache is cached in the client. Send it once to the server and it's stored.

8. Sharding is used to get more write performance.
    1. Writes are 60% workload. Usually it's 10:1 the other way. They still use MySQL for data storage because they are very comfortable with it's performance statistics under load, etc.
    2. For each shard there's a master and at least on read replicas. For most there's just one read replica, but it depends on the access pattern of the service. Reads are split to the read replicas. For a few places that do have more reads they have more read replicas. Read replicas are also used to keep data remotely as a backup.
9. Playfish is driven by pure necessity. They built their own key-value store because it had to be done. Why not use NoSQL? They are looking into the options, but at the same time they have a solution that works, that they know how it will behave. Interested in NoSQL solution for the operations side, for managing multiple databases. It wasn't easy to go into the mode of running NoSQL, but it was a necessity driven by their requirements.
10. In a scale-out situation you have to go to something like sharding and at the point many SQL features go away. You have to do a lot more work yourself. Now you just can't add an index when you have blobbed and sharded.
11. When going NoSQL you are giving up flexibility of access patterns. Relational databases are good because you can access the data in anyway you want. For example, since they can't use SQL to sum up fields anymore, they both aggregate on the fly or they use a batch process to aggregate.
12. Backup is to S3.

Flash - The Client

1. Client side CPU and resources scale with the number of users so it's sensible to make use as much as possible of the client. Push as much processing as possible to the client.
2. Changes are written asynchronously back to the server, which helps hide network latency from the user.
3. Changes are checked on the server side to detect cheating.
4. Flash talks to the Java application servers using a service level API.
5. Bringing processing closer to client gives the user a better experience. A website brings servers closer to users. Playfish brings the processing even closer, on the client.

YAMI4 - Messaging

1. YAMI4 is the messaging system Playfish decided to use after a long evaluation process. It offers point-to-point connectivity, low latency, no single-point-of-failure, and asynchronous messaging for event-driven processing and parallelism in multiple backend services.
2. After services go through a discovery phase to learn where each endpoint is located, messages are transported directly between services. It's a brokerless model. Messages aren't funneled into a centralized services and then redistributed. Messaging is very efficient with this model, latency is reduces because there are no intermediary hops. They specifically did not want a separate component to handle traffic and then transfer messages around. This approach reduces failure points and latency.
3. Considered Thrift, but YAMI4 won because of it's asynchronous operations. Thrift uses a RPC model that looks like a local function call, which makes it harder to deal with errors, timeouts, etc. YAMI4 is a messaging model, not a RPC model.
4. YAMI4 doesn't handle the service discovery aspect, they built their own on top that does a discovery phase and then talks directly to the other service. It's more how the Internet works.

5. As a messaging system, messages do not invoke methods on objects. No objects flow over the network either. Objects live in the services. Each service is responsible for activating, passivating, and dispatching operations.

Dealing with Multiple Social Networks

1. One of their main challenges is to be able to support so many different and increasingly, different types of games.
2. The principle of *loose coupling* is used as much as possible:
    1. Team structure is matched to architecture by services being owned by teams.
    2. Services keep well-defined interfaces, so that each team can iterate on and deploy their own service without affecting other teams.
    3. When interfaces need to change, interfaces are versioned. They try to maintain backward compatibility so other teams do not have to roll out changes.
3. To facilitate all of this, a common set of standards is applied to all services:
    1. Common service transport (YAMI4)
    2. Common operational standards such as how services are configured and how they provide monitoring information.
4. A combination of common standards and loose coupling allows both development and operations teams to be agile and efficient.

Development and Operations

1. Services are released independently from each other.
2. Resources are separated by service. A problem with one service will not impact an unrelated service.
3. Teams are organized by service, although there is some overlap.

4. Operations teams are separate from the development teams, though there is close relationship. No big handoff phases. They don't just throw a release over the wall. Operations is included in design.
5. Developers don't release code. They check it in and operations picks it up.
6. Teams have a lot flexibility in how they deploy. Most games have a weekly release cycle. Everything is iterative, which means the code is good enough to go live, but features are not necessarily completely finished. The weekly release cycle is especially valuable for games with virtual items because users like to know there's new exciting stuff every week. Other teams can work in longer feature chunks. It depends. That's one of the advantages of working on SOA. Teams can work independently. There's doesn't need to be one release cycle.

Java - The Server

1. Java supports creating clean reusable components.
2. There are a wealth of open-source libraries.
3. Java is flexible: it can be used to implement web applications, process requests, batch processing, and event driven systems.
4. Java has many tuning options. They've done a lot of work tuning garbage collection to hone performance.

Game Design

1. To make games users will enjoy Playfish combines data driven design with good old human inspired game design. Data is used to inform a lot of decisions about making games that users will enjoy. They can see from the data what users do the most. That tells them how to make and individual game better, but also what to do when designing new games.

2. Playfish loves behavioral analysis. They look at what people are doing inside games and then asks them why. In support, the client and server are heavily instrumented to generate events on user actions. These events are then processed to create an aggregated information about what the users have done in the games. They are now using EMR/Hadoop/Hive to be able to access this huge collection of data with a SQL like interface. Using EMR in cloud is working stunningly well, they are very happy with it. Huge amounts of data can be stored in a granular form and it's still possible to quickly find out what you want because everything is designed to work in parallel, which makes an excellent fit for the event type data produced by games.
3. It's quite surprising what users will want. They may not even know it themselves. What people think they want is different than what people actually use. Sometimes costly features are put in that users say they really want, but they end up not using them. Then there are features that nobody mentions but people use all the time. People who post to forums are often people who dislike stuff, so it's easy to get a very unbalanced view because they are really passionate about the game, they hate every change. Data helps find out what people really love.
4. Game design can't be just data driven or games will be soulless. You have to get the data driven balance right. You can't just do what data says do. Go with your inspiration about what to add or change in a game, then use the data to help polish it up.
5. Some features take a lot of work to create, but users don't end up them because the features are too complicated. They setup funnels to see if people abandoned features before completing them, then they can figure out why. Maybe a feature needs to be tweaked or abandoned. Engagement is tracked as a measure of the love users have for a game. They then invest in content that keeps people coming back so the ecosystem will grow.

6. Agility + Data + Design = allows new inspired features to be tried out quickly to see if they work or not. The result is a funner game.
7. Teams are organized into small close-nit groups that have full creative freedom. All team members get a say on what makes a game more fun.
8. Experiments with games are run to go after a niche to see if the niche responds. Some will be successful, some less so.

Lessons Learned

1. **Build an architecture that takes advantage of the special nature of your application.** Playfish has tailored an architecture to their very specific needs. Don't try to build a general architecture that will scale everything. Take advantage of the nature of your space to make life as easy as possible.
2. **Don't worry about getting it right first time**. Get something out the door and start learning. If they tried to get where they are now 3 years ago, they would still be building the system. They got something out the door that couldn't scale at all, but they got it out in 5 weeks. This is the key.
3. **Don't be afraid to stick with things you know and understand**. Pick stuff you are familiar with. You don't need to go with the newest and coolest. There's a lot of value in knowing a product well and being able predict how it will operate on your use cases. You will get into a lot less trouble that way.
4. **Keep things simple first then scale when need to**. Use that lead time to build out what you need.
5. **Shard and BLOB**. Use sharding to scale out writes. Use BLOBs to reduce the number of records per user to one. This speeds up object access and allows more objects in RAM.
6. **Always have rough idea how you are going to scale**. Don't design features too early, but don't create disruptive dependencies. For

example, it's fine to launch without scaling in place, it's not cool to have a feature that isn't going to work after you shard.

7. **Scaling data is harder than scaling processing**. Where are you storing data? How many people need to read and write it? Stateless app server make it easy to add more processing, scaling the data is a lot harder.
8. **Don't over complicate things**. Scaling a simple system is easier. Keep as simple as possible for as long as possible. This allows you to learn where your pain points are and then you can address those specifically. A lot people think they have to build the huge thing from day one. For example, if you don't need multiple levels of caching then don't put it in, because you have to support it.
9. **Use SOA manage complexity**. As new games are added splitting code up into different components that are managed by different teams helps keep the overall complexity of the system down, which helps make everything easier to scale.
10. **Take calculated risks**. Playfish went to the cloud but had a backup plan in case it failed. It was a risk, but it also had a lot of benefits. If Amazon pulled their service then if they really had to they could move. That's the benefit of using Infrastructure-as-a-Service. With Platform-as-a-Service there's a much greater risk because you build a lot deeper dependencies. For example, the risk o fPlayfish moving to a NoSQL product when they have their own working key-value database is also too great.
11. **Operations and development should understand how a system will work in production**. Is it easy to manage? Is it easy to support customers? How will it be configured and monitored? Develop what you need to make all this happen. Developers can't just throw code over the wall. There should be no wall.
12. **Use data to help find out what people really love.** Users don't always know what they really like best. Instrument your code and analyze the data to find meaningful patterns and use that

information to continually improve the system.

13. **Hardest thing to scale is people**. Have to make it really easy for people to do their work. It's a lot easier to get a lot more servers than it is to get a lot more people.

I'd like to thank Playfish for taking the time for this informative and insightful interview. If you would like your architecture featured on HighScalability.com, please contact me to get started.

Playfish is looking for people. For career opportunities please see their Jobs Page. From my dealings with a few of the folks a Playfish they seemed to have their stuff together. Worth a look as they have many open positions and they are looking for server side engineers. They want to build new systems to understand users better using EMR/Haddop/Hive and scale the overall system to add more users and games.

Related Articles

1. Playfish Shows How "Games-as-a-Service" Scale in the Cloud
2. Interview: Playfish On Finding Meaning In Simple Actions With My Empire
3. AWS Case Study: Playfish
4. EA Playfish exec touts next-generation Facebook games
5. An In-depth Look: Playfish
6. Introducing Playfish
7. Lives Notes from “Asynchronous Games on Social Networks” at Social Gaming Summit
8. Video: SGS09: Building Social Games At Scale.
9. Video: SGS2008: Asynchronous Games on Social Networks.
10. Asynchronous Games.
11. ASYNCHRONOUS MULTIPLAY: FUTURES FOR CASUAL

MULTIPLAYER EXPERIENCE by Ian Bogost.
