ESPN went on air in 1978. In those 30+ years think of the wonders we've seen! When I think of ESPN I think of a world wide brand that is the very definition of prime time. And it shows in their stats. ESPN.com peaks at 100,000 requests per second. Their peak event is, not surprisingly, the World Cup. But would you be surprised to learn ESPN is powered by only a few hundred servers and a couple of dozen engineers? I was.

And would you be surprised to learn ESPN is undergoing a fundamental transition from an Enterprise architecture to one capable of handling web scale loads driven by increasing mobile usage, personalization, and a service orientation? Again, thinking ESPN was just about watching sports on TV, I was surprised. ESPN is becoming much more than that. ESPN is becoming a sports platform.

How does ESPN handle all of this complexity, responsibility, change, and load? Unlike most every other profile on HighScalability. The fascinating story of ESPN's architecture is told by Manny Pelarinos, Senior Director, Engineering at ESPN in the InfoQ presentation Architecture at Scale at ESPN. Information from Max Protect: Scalability and Caching at ESPN.com has also been folded in.

Starting in a pre-personal computer era ESPN developed an innovative cable and satellite TV sports empire. From an initial 30 minute program reviewing the day's sports, they went on to make deals with the NBA, USFL, NHL, and what would become the big fish of all sports in the US, the National Football League.

Sport by sport deals were made to bring sports data in from all possible sources so ESPN could report scores, play film clips, and generally become one stop shopping for all things sports on TV and later the web.

It's a complex system to understand. They have a lot going on with Television & Broadcasting, live scoring, editing and publishing, Digital Media, giving sports scores, web and mobile, personalization, fantasy games, and they also want to expand API access to 3rd party developers. Unlike most every profile on HighScalability ESPN has an enterprise heritage. It's a Java Enterprise stack, so you'll see Oracle databases, JMS brokers, Java Beans, and Hibernate.

Some of the most important lessons we'll learn about:

Platform changes everything. ESPN sees themselves as a content provider. These days content is accessed through multiple paths. It can be on TV, or on ESPN.com, or on mobile, but content is also being consumed by more and more internal applications, like Fantasy Games. And they also want to provide an external API so developers can build on ESPN resources. ESPN wants to become a walled garden built on a sports content platform that centralizes access to their prime advantage over everyone else, which is unprecedented access to sports related content and data. The walled garden approach that Facebook has made work for social, Apple has made work for apps, and Google has made work for AI, is what ESPN wants to do for sports. The problem is transitioning from an enterprise architecture to a platform based on APIs and services is a tough change to make. They can do it. They are doing it. But it will be hard.

Web scale changes everything. Many web properties today use Java as their standard backend development environment, but ESPN.com, which grew up in the Java Enterprise era, went all in for the canonical Enterprise architecture. And it has worked quite well. Until there was a sort of phase transition from enterprise class loads experienced by a relatively predictable ESPN.com to a world dominated by high mobile traffic, mass customization, and

platform concerns. Many of the architecture choices we see in native web properties must now be used by ESPN.com.

Personalization changes everything. The cache that once saved your database is now much less useful when all content becomes dynamically constructed for each user and must follow you on every mode of access (.com, mobile, TV).

Mobile changes everything. It puts pressure everywhere on your architecture. When there was just the web architecture didn't matter as much because there were fewer users and fewer servers. In the mobile age with so many more users and servers these kind of architecture decisions make a huge difference.

Partnerships are power. ESPN can create a walled garden because over the years they have developed partnerships that gives them special access to data that nobody else has. It's good to be firstest with the mostest. That individual sports like the NFL and MLB seeking to capture this value with their own network lessens this advantage somewhat, but the forces are such that everyone needs to get along, which puts ESPN in the middle of a powerful platform play, if they can execute.

Stats

Internet's #1 sports website. Top 10 of all sites. 5th largest website among men ages 18-54 (Facebook, Google, Microsoft, Yahoo are bigger).

Powered by only a few hundred servers.

A few dozen serve the major portions of the site, like the front-page service.

Only a couple of dozen engineers.

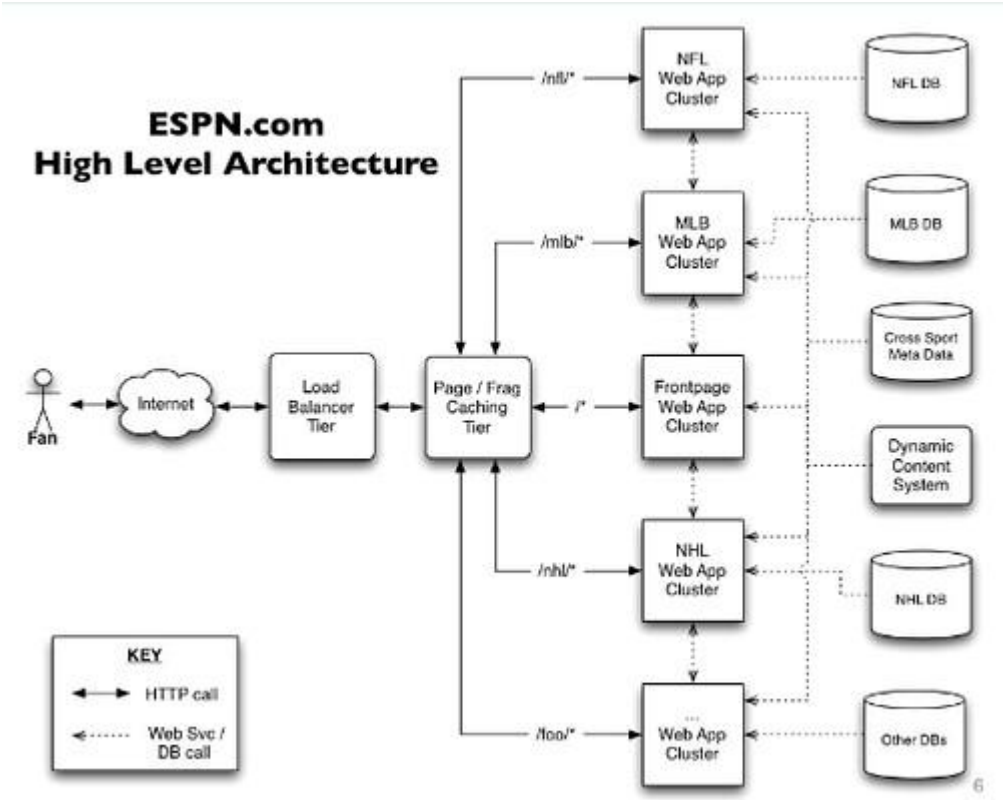
Peak 100,000 requests per second. Peak event was the World Cup.

Sports specific data is gigabytes in size.

Stack

- Java based.
- Oracle database,
- AQ Streams
- Message Broker
- WebSphere MQ
- Interesting integration of people on the ground as sources of data as well as automated feeds
- JMS Broker
- Microsoft SQL Server
- Hibernate
- Ehcache
- IIS
- IBM eXtreme Scale
- Mashery
- F5 Load Balancer

Architecture



Started over a decade ago with Starwave, a Paul Allen startup company, so they got a good deal on Microsoft tech, but selected Java. Java was still young so they had to roll most everything on their own. The site has evolved over 100s and 100s of iterations.

Expansion has been through more services, Watch ESPN, for example, is a new dedicated service. Over 100 logical databases and over a couple hundred of distinct applications deployed over hundreds of servers.

Goal of ESPN.com is to serve sports fans anytime, anywhere, on any device, with accurate and timely data, with access to deeper content. Scores, stats, and deeper content must be accurate and immediate. No outages, just like TV.

Do not consider themselves to be a technology company. They are a media and content provider. Owned by Disney, not publically traded.

Digital Properties: ESPN.com, Fantasy games, Mobile (WAP, iPad, iPhone, etc), WatchESPN (watch cable on your phone) , ESPN the Ocho (Deportes, W, HS, etc). Different architecture and services power each property. ESPN.com is what is primarily covered in this talk.

Different local sites in say Boston and New York. A large team of editors provide local twists for each local site so different sets of fans will get different slants on the game.

Key to handling high loads with a low hardware footprint is a sophisticated page and fragment caching system. Live updates like scores, stats, schedules, have a different caching system. Personalization also has its own caching system.

Developers generally do not have login access to production machines.

Architecture is Organized Around Applications and Databases

- Dozens and dozens of logical databases. A database for MLB, NFL, NHL, etc., and applications for each including more abstract services like Frontpage for servicing the main website.
    - Process isolation. If faulty code is deployed it won't take down other parts of the site.
    - Not a monolithic architecture, there are different systems for different sports.
    - Historically primitive SOA model. Frontpage makes several HTTP calls to different services that return XML and the caller parses for what they want. Lots of interconnections between different services.
        - Frontpage scoreboard has all the different scores from all the different leagues. A separate application for each sport. Click on NHL hits a VIP through a load balancer that takes you to a NHL application.
        - There's frontpage app that services frontpage widgets by making calls to other services.
    - Application service. In front of the databases is web application server with all the business logic for the sport.
        - Usually just one application service per sport.
        - Applications are clustered so that are 6-10 application instances in the cluster so they scale out horizontally
            - When football is in season, for example, the number of application instances is scaled up and the back down as seasonal demand dictates.
            - Disney datacenter has some elastic capabilities, but looking at Amazon for improvement in this area.
- All feeds flow into Sports Data Repository (SDR).
    - A really big Oracle database.
    - The same stats you see on the .com site are the same stats used to create the bottom line ticker on TV and some of the panels on Sports Center.
    - TV access stats using web service calls.
    - Television & Broadcast doesn't need to scale in the same way as the other properties so they can consume data directly from the Oracle database.
        - Oracle AQ streams is used distribute messages on the Oracle side.
        - Message gateway distributes messages to the Digital Media side, which has its own JMS Broker.
        - Housed in Bristol Connecticut.
    - Digital medial, like ESPN.com, is fronted by systems that create feeds and normalize messages for consumption inside the enterprise.

        - JMS Broker (WebSphere MQ) is used to distributed messages.
        - Housed in Las Vegas Datacenter.
    - .com and TV are in different datacenters. 80 milliseconds of latency between the two datacenters. The two JMS brokers are highly optimized send updates as fast as possible.
    - PubSub is the architecture above the JMS brokers. Applications listen on different topics, read messages off the queue, unmarshal into JavaBeans, persist into the database, then push that data directly into the web app server.

Where do Stats Come From? (Data Ingest)

- For each database there's a batch processing or feed processing service that is responsible for updating all stats, schedules, standings, scores, etc.
- Most stats come from 3rd party vendors or the pro leagues themselves.
- Direct feed. They have a partnership with MLB, for example, and data is sent directly from MLB. Gives them a huge latency edge over other services.
- Stadium feed. Pitch effects (pitch location) data is sent from the stadium.
- Manual feed. ESPN personnel score the game and feed data into the system manually.
    - College football doesn't have a stats feed so data is entered by people watching the game.
    - Failover. Also acts as standby failover system in case something happens with the automatic feeds. The game can be taken over manually is there's a problem.
- Almost all stats are overwritten nightly with "official" stats from a 3rd party. Lots of tools and systems in place to make sure feeds are accurate, but corrected data comes in at night.
- Relatively low message rates except when games are going on. Complicated because all game events need to be processed in order.
- Same stats that power .com also power TV, but not accessed in the same way.

Application Services

Services talk to each other with a direct call via a local service, remote EJBs, or REST.

Preferred mode inside a datacenter is EJBs for which there's sophisticated caching layer. Accessed from Java clients.

For REST services there's a simple TTL based caching layer. Accessed from PHP, Javascript, etc.

On the Digital Media side they use Microsoft SQL Server and on top of that Java Application Servers. It scales because they try not to touch the database. They cache everything so the database isn't hit.

Application Level Caching

Historically, object caching in the web application keeps an in-memory hash map of table objects. Want a game it's cached forever in the hash map until it fills up or is expired via a database trigger. The cache is replicated per application server.

This approach is simple and worked before the proliferation of mobile devices as it was relatively easy to predict the number of servers necessary to serve NFL traffic, etc. With so much mobile traffic there's a stampede of expire messages going on. As games stats are updated, for example, expires are going out and all the servers are asking for updated game scores. Changes are now pushed instead of expired.

Page Cache Framework

- High performance page and fragment caching.
- Caching is another homegrown technology called Stout. Uses IIS web server with an ISAPI page caching plugin. Written in C++. Runs on cheap low end hardware so it extremely cost effective. Uses TTL caching exclusively. See 1000+ requests a second. The app tier received 100+ requests a second. The app tier has its own caching layer. Dozens of Stout servers protect the app tier. The app server tier protects the database tier so it hasn't been necessary to horizontally scale out the database. Stout pulls out bad app servers from rotation. Looking at Varnish which appears to be faster.
- Cache is King. Most important part of the architecture is the page caching tier. Heavy use of page caching where possible.
- Per URI, TTL based expiration. Transparently talks to the load balancer and says for certain URIs we want to cache for a certain period of time.
- Highest accuracy is a low TTL and block until it returns with data. Slowest access. Used for scoreboard data, for example.
- Lowest accuracy is a high TTL that doesn't block, it returns dirty data. Fastest access, used for data not updated frequently. Use for schedule data, for example.
- Editorial content and other content that doesn't change frequently can be cached with a longer TTL.
- Automatically demotes unresponsive servers.
- TTL based caching doesn't work very well because it causes the database to be hit with frequent accesses. Imagine a TTL of a few seconds with dozens and dozens of servers, it doesn't work well.
- Saved millions of database accesses per second at peak times per sport. Huge win. When there was just the web all this didn't matter as much because because there were fewer users and fewer servers. In the mobile age with many more users and 50+ servers these kind of architecture decisions make a huge difference.
- Advantage of their custom solution is that it runs on cheap / low end hardware.
    - 100s of thousands of request to load balancer
    - 100s of requests to actual app server
    - 10 MLB servers instead of 50 means big savings

Common Object Model

Even though they have many different databases corresponding to different sports, in front of it they have a common object model. Put as much as possible that's common across sports in the common model.

Every game has teams, two teams usually. Olympic sports have competitors. Code is the same in the common feeds which allows powerful operations like get me all game scores today, regardless of sport, for an all sports page.

When in a specific sport they have sports specific models. One for NFL, MLB, etc. When they have a detailed what happened today page on a sport they use the sports specific model.

A tier hides the complexity then they fall back to a sports specific model when necessary.

Hibernate

- TTL caching doesn't work because data doesn't change that often during the day, but often during the game.
- Rarely are lookups by identifier, they are mostly by query, like give me the number of games going on for the whole week so a schedule can be shown, or show me all games where a particular team is playing. Hundreds and hundreds of queries.
- Easy to use at first. When things get more complicated or you need optimal performance it is very hard to use efficiently.
- Ehcache is used as a second level cache for Entity updates and works great.
- The Hibernate query caching mechanisms are not sufficient so they built a JPA Query Replicator.
    - State changes frequently (say a game score), but the change is is the same for everyone. Doesn't work for personalized data or fantasy, works when blasting changes to all users.
    - On the process that's getting the updates, a listener on the entity side they watch for any updates they care about using a rules engine. Queries are filtered out based on what was updated. An update for a particular game should not trigger a query that asks for data since the beginning of time. The query is rerun and pushed out the the cluster so all users are updated which takes heat off the database.

Content Management System

UI not so good, the focus is on performance and scale.

Two subsystems: CMS and DCS (dynamic content system).

The CMS for editorial staff. Highly optimized for querying. Optimized UI for users that make changes to stories. Full relational database model (SQL Server). Only a few hundred editors so it doesn't need to scale horizontally.

The DCS uses SQL Server, but is denormalized and stored as a blob type. Retrieval is fast. Edited data goes on in the CMS. When an article is published the content serialized and put into the DCS. DCS is a cluster of 10 servers that can be horizontally scaled.

All 76 services (MLB, NFL, etc) have a plugin that knows how to talk to the DCS. The plugin also has a cache. So on a publish an expire is sent to each client so it will pick up new content from the DCS.

Templating engine built by Disney Internet Group. Disney owns ABC and ESPN. A decade ago built language called T. Very high performance. More like JSPs than JSF. Have build a hundred thousand templates over the years.

Cheap on hardware side so they have to be efficient on the software side which is why they haven't moved to other slower templating systems.

Live Scores (ESPN.com)

Most content doesn't update that quickly. Editorial content doesn't update that quickly. For scores, they want to have the fastest scores possible, so it's treated differently on the front-end and backend.

Master feed template that has all the data for a game. A process polls the template for updates and puts the changes in a snapshot table in the database. Something called Caster sits in front of all that. It's custom technology that's like web sockets before web sockets existed. There's an open socket connection from the front-end to the back-end, which is a big cluster of Caster servers, and updates are streamed over the connection. On the front end there's a flash connector. A very high end server does nothing but manage connections to the front end flash connector. It can do 100K+ concurrent open socket connections. Every time a snapshot gets updated it is sent over the proper connection. JavaScript running on the page reads the updates and displays it on the page.

Flash connector downgrades to polling at 30 or 60 second intervals. Terrible for bandwidth usage. Looking at web sockets as replacement.

Personalization

- Personalization is you telling them what your favorite things are: sports, players, teams, fantasy data. These are things specific to you.
- Quite different than the rest of ESPN.com and has a lot of specific requirements.
- Difficult to cache because it's 1-1 instead 1-many as with other caching schemes. And the data must follow you everywhere (.com, mobile, TV)
- Have a lot personalization data. Over 500GB. Doesn't fit into a single JVM or a database that scale to the throughput they need of 10s of thousands of requests per second.
- Built a data grid using IBM eXtreme Scale.
    - It's a distributed in-memory hash map. Bought 7 servers with 96GB of RAM each. Software automatically balances and partitions data to JVMs.

    - Big problem with JVMs is garbage collection, so they have 100s of JVMs running per server to minimize garbage collection and the software shards everything automatically.
    - Only store stuff unique to you. If you are a Yankees fan they just store the ID of the Yankees, not all data about the Yankees.
    - eXtreme Scale was harder to setup than Coherence, but cheaper and they got more support from IBM than Oracle.
- A Composer system knows how to talk to all services like NFL and MLB and it knows how to talk to the grid. So to build a personalized page they'll go to the grid and go the correct service for the teams you like and build a schedule, scores, fantasy data, columnists, etc as a JSON feed. On the client side the data is assembled. Handles 10s of thousands of requests per second with 6 commodity servers for Composer.

Fantasy Games

Very different use case. Fantasy data is calculated from real stats that are then transformed into "made up" data, so they have different ingest processes.

APIs

APIs for editorial content. Rights for data is tricky, but they own the content so that's what they've released.

One of the biggest problems they have is recruiting and onboarding new people. Having documentation and consistent APIs will help outside the firewall with developer APIs but it will help internally because it will be easier to build applications.

Some of the tricks with the EJB tier can't be applied to the APIs so it's not all figured out yet.

Figuring out APIs is hard. Developers want all data in one call. But they are leaning towards finer grained APIs that would require many more calls and more work from the developer.

Mashery is used to protect the API using TTL caches. Different levels of polling. External users are throttled to so many requests a minute. Internally the limit is less restrictive.

The most complicated sport is soccer because there are so many feed providers that they have to work with to get the data. The SDR team normalizes all the feeds into a soccer feed. If something goes wrong with a feed they can take over a feed and manually add it to the system.

Special Effects

From ESPN Emerging Technology's use of NVIDIA'S GPU Solutions for High Resolution Imagery. Not a lot of details, so these are basically just bullet points from the slides, but it's cool stuff and looks like magic on the screen.

ESPN is an innovator in advanced real-time graphics and they are using GPUs for high value features like Huck-O-Meter, HRD Ball Track, Snap Zoom, Ref Mics, Sky Cam, Ultra-Mo, Player Tracking, the 1st & Ten Line, K-Zone, the Emmy-winning EA Virtual Playbook and much, much more

Each GPU in the system is classified as either an Input, Output, Input / Output, or a Compute Engine

All GPUs have peer-to-peer access via CUDA

Multiple input cards and output cards may be assigned to a GPU

Hardware abstraction layer allows video I/O hardware from several manufacturers via gpuDirect

Each GPU pipeline can handle unique video formats for input vs. output

The Future

- Have a common API for all data with sport specific extensions where necessary.
- Moving to a cache push model (don't expire). With mobile and a horizontal model and more and more servers more servers are going to the database for updates. As they don't have big beefy databases this becomes a bottleneck. If scores stop updating on NFL Sunday it's not a good day.
    - As data comes in it is marshalled into a common object model format. It's asynchronously persisted to the database and also asynchronously pushed to the rest of the cluster. Instead of 6 servers, for example, going to the source when something changes it's pushed directly to the consumers and they don't have to go to the database.
- More loosely coupled services. Locally (same JVM), via Java remoting, REST.
- Create one service for all sports.
- Leverage generics and code gen to expedite process converting the rest.
- Caching everywhere to protect all components from load.
- Deliver more personalized content and have it follow you everywhere (.com, mobile, TV).

Lesson Learned

Cache push model. As data comes in asynchronously persist to the database and also asynchronously push to the rest of the cluster. Changes are pushed directly to the consumers which means the consumers to stampede the database to pick up new changes. Not necessary in a more static world of predictable resources, but key to scalability in a mobile world.

Good enough is good enough. When you are starting early in a technology you'll have to build a lot yourself that will later look pretty silly as new and better code comes out. But you need to code and make it work, so it's good enough.

You can do a lot with a little. You think about ESPN and you think industry leader. Yet ESPN uses remarkably few compute resources and few developers to create a high value, highly visible system. They think about efficiency. Whereas a company with a web heritage might just add machines as needed with the excuse that machines are cheap, ESPN actually thinks about saving money to make a profit. A strange concept.

Know your audience. Decisions are driven by their goals. Devices everywhere, fast accurate data, broad coverage, and high availability. Those values are reflected in the architecture and the decision making process.

Manual failover. An interesting aspect of their system architecture is that includes both manual input of game data but also manual failover policies when the automated feeds fail. Probably not something most people think about as an option, but it shows a high dedication to their goals of having fast accurate data.

Tailor systems for different use cases. They let the requirements of the different sports and services drive the architecture. This allowed for parallel development and just getting stuff done. For example, they built a query caching mechanism that specifically optimizes for the case of game updates and distribution because that's their business.

Make different things look the same. While the let a thousand architectures bloom approach is great in times of great change and development, it sucks when you want to make a common API service layer or make system wide optimizations in response to stressors like mobile or personalization. So the opposite force is to make a common architecture, a common data model, and then fall back to type specific models when necessary.

Cache to protect the database. Not a new idea at all, but this was a core scalability strategy that worked quite well for ESPN. This allowed them not to invest a lot in the database tier. But personalization, which is the wave of the future, is a cache buster.

Consistency helps all developers. Having documentation and consistent APIs helps outside the firewall with developer APIs, but it also helps internally because it will be easier to build applications.