Scalability and Robustness in the Domain of Web-Based Tutoring

Jozsef PATVARCZKI, Joseph POLITZ, Neil T. HEFFERNAN
Worcester Polytechnic Institute
100 Institute Road, Worcester, Massachusetts 01609, US
{patvarcz,jpolitz,nth}@cs.wpi.edu

Abstract. Web-based Intelligent Tutoring Systems distribute content using a server based infrastructure. These systems need to be easily accessible to provide a reliable and scalable architecture for students, teachers, parents, content creators, and researchers. Scaling up web-based applications requires a well designed system that distributes the load accross multiple application servers and database server(s). A typical web application involves a 3-tier architecture where client machines, application servers, and database server(s) are working together to achieve better performance. In this paper, we describe our architecture that utilizes these techniques to form a scalable, reliable, and robust structure. Our solution is general, and can be applied to solve different scalability problems in the domain of Web-based intelligent tutoring systems.

Keywords. Scalability, robustness, intelligent tutoring systems

1. Introduction

The importance of user interactions with a web-based Intelligent Tutoring System is significantly increasing. The idea that a user interacting with an online system that manages all activities can create a performance bottleneck was suggested by Ritter and Koedinger early in 1996 [9]. In the case of web-based tutoring, accessibility is one of the key factors. Students, teachers, content creators, and parents can get access allowing them to do many things including getting reports on a student, creating new content, solving homework assignments, and managing classes. Building intelligent tutoring systems is generally agreed to be much harder, Anderson [4] suggested that there is a cost of at least 200 hours to build an intelligent tutor for each hour of instruction. With a web-based system, we can eliminate the cost of installing software on client machines and the cost of information collection with a centralized environment. Brusilovsky described that web-based systems have longer life cycles than client-based [5]. Nowadays, as the number of the available Internet accesses is increasing, web-based systems have more potential compared to client-based ones. For example, one of the components for content creators is the builder. The builder allows a user to create pseudo tutors composed of an original question and scaffolding questions which break down the original one into different steps. In the case of a 24/7 system, this component is always available and content creators can interact with the system continuously with greater control over content distribution. The increasing demand for an always online intelligent system generates new set of problems. The performance requirements, scalability needs, distributed components, cache management, etc. will be active parts of an effective web-based Intelligent Tutoring System. This paper describes our solutions for scalability and performance problems in the domain of Intelligent Tutoring Systems.

2. Scalability factors

2.1. The Assistment system

Assistment [7] is an intelligent web-based tutoring system that has had success at tutoring middle-school math students in central Massachusetts. Emphasizing the types of questions seen on the statewide MCAS test, Assistment has a number of users (students and teachers) who use the system every day in class.

While the Assistment system hosts a total of around 3000 students, the average number of active users varies from around 200 to 500 on a given school day. Most of the load comes from 20-25 classes distributed among roughly 10-15 schools. Assistment offers constant (24/7) service to all of its users at these locations. This availability is important because Assistment aims to extend the reach of the ITS outside the classroom to out of class work, homework assignments, and independent learning. Being a web-based tutor allows the system to support these on-line activities - there is no special software required other than a web browser, and the full experience is available anywhere with an Internet connection.

The current architecture of Assistment is more than enough to sustain this level of use from a performance standpoint. However, the goals of the project extend further than just maintaining the current levels of use. There are 337 middle schools in Massachusetts (http://massachusetts.schooltree.org/middle/counties-page1.html), and it is the goal of the Assistment team to be able to support students at all of these schools consistently and stably. This represents an order of magnitude increase in the number of expected users. We have proposed a new architecture based upon and improving upon the principles that were used to create the current successful system.

2.2. Current Architecture

The current architecture of Assistment contains 4 application servers, a database server, a load balancer, a web cache, and a machine hosting a network file system shared by the application servers. The application is served on each server through multiple single-threaded Mongrel [10] instances. Five Mongrel instances run on each application server for a total of 20 live server threads to which the load is distributed by the load balancer, which runs separately from the application servers themselves.

2.2.1. Software Implementation

The Assistment application is written primarily in Ruby [6], using the Ruby on Rails [12] framework for web applications. Javascript is also used, less importantly to provide extra functionality to the views of the application, and more importantly to allow computation to be done within the client's browser as opposed to on the application server.

Ruby on Rails is an especially appropriate framework for this kind of application. It allows for rapid prototyping of new features, which allows it to support many and diverse requests from users. It has good support for incremental development towards full-fledged implementations of the desired end features. Finally, it supports a fast-paced development environment with a changing development team. Assistment is developed entirely by students, both graduate and undergraduate, and the design and development team can change dramatically over the course of an academic year. Having a framework that supports this sort of documented incremental development makes transitions in development teams much easier.

The implementation of the tutoring part of the application in Javascript was made for practical performance reasons. In web-based applications, there are two models of distribution of responsibility between client and server - the client can be said to be *thick* or *thin*. A thin client does little or no application computation, and is primarily a terminal and a view of the application that runs on the server. This is true of most online stores, for example. A thick client is responsible for a lot of computation and application logic, and mostly communicates transactions of persistent information with the server. Most graphics-intensive online multiplayer games are good examples of thick clients. Providing the tutor to the client in Javascript allows the application to distribute the computational responsibilities of scaffolding logic and determining correctness to the client. Javascript also has the benefit of only requiring a web browser to run - the client is not required to download any special software to use the application.

2.2.2. Application Servers and Load Balancing

Assistment serves content using the web server Mongrel. Mongrel is a single-threaded server, so multiple instances are run to acheive concurrency. To distribute the concurrent load, Assistment currently runs 5 Mongrel instances on each of 4 physical application servers.

To balance the load amongst these application servers, the lightweight http server nginx [11] is used. Nginx can efficiently balance the load among the 20 Mongrel instances. Another important feature of nginx is automatic failover. That is, if a given application server (or even a single mongrel instance) goes down, nginx will skip over requests that fail to these servers until a successful connection is made to a different instance in the application server pool. Unless multiple servers are inactive at once, this should lead to no noticeable difference in user experience. This kind of setup has several distinct advantages for a system that aims to have constant availability. A single server can be serviced, upgraded, added, or removed without the application having to shut down. This kind of robust, failover behavior is key in designing successful, scalable architectures.

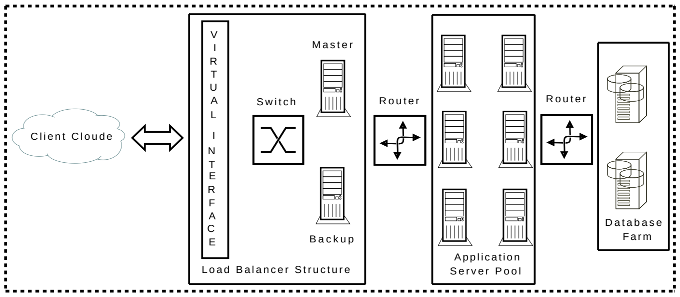

Figure 1. New architecture of the Assistment system

2.3. Robustness and Scaling Up

2.3.1. Robustness

Once a web based tutor (or any web based application) has been established as effective and usable, robustness becomes a key issue. A significant amount of investment is put into the system and application, and protecting its integrity and maintaining its availability are important goals.

A system component is a single point of failure if the rest of the system cannot operate without it. In the existing system, for example, the database server is a single point of failure. If the connection to the database server is lost or there is a hardware failure in the machine, the entire system ceases to function. In contrast, a single Mongrel instance or even an entire application server could fail, and the load balancer would handle routing requests to the other application servers.

A robust system has as few single points of failure as possible, and a suitably robust system can be resilient to multiple failures. There is no rule that says whether a given single point of failure is easy or difficult to remove. In most cases, it is simplest to design redundancy into the system from the start to avoid these single points of failure.

We propose a new architecture for Assistment, shown in Figure 1, to address robustness concerns in the system while scaling the system up considerably. This architecture contains of three major parts. The first part recieves the requests from the client cloud through a virtual interface. This interface is responsible for routing the incoming requests to the master load balancer and switching to the backup one in the case of a master balancer failure. This automatic replacement eliminates the single-point-of-failure problem at the frontend. If the master balancer recovers then the interface switches back to the original state. For the virtual interface, the Assistment system uses the Common Address Redundancy Protocol [1] (CARP). The master load balancer routes the incoming requests to the second component of the system and it allocates an application server from the pool. The application server pool has an increased number of Mongrel servers compared to the existing system, and maintains the connection with the PostgreSQL [8] database farm through Pgpool-II [3]. Pgpool-II acts as a router between the second part of the architecture and the backend. It can be used to manage multiple database servers and fully replicate writes between nodes. This middleware has the ability to distribute select queries between the two databases in a round-robin fashion.

Furthermore, it supports robustness with a built in automatic failover detection in the case of a database problem.

In the upcoming sections, we will consider a number of issues relevant to the design of a robust and resilient system.

2.3.2. Models of Load

Web-based ITS encounter a somewhat distinct type of activity from their users. The type of traffic that comes during class time can be much different than the type of traffic when organized classes are not in session. There are two predominant types of traffic that a web-based ITS will typically see.

The first model is a rapidly changing, heavy load that will occur when there are multiple concurrent classes at multiple schools. Here it is crucial to handle multiple, rapid requests for logins, problem set (content) requests, and logging of student progress. Major concerns here are performance of the load balancer to distribute requests, accurate logging of progress, and serving large amounts of content rapidly.

The second model generally applies to non-school hours, when access is more sporadic. Teachers often generate requests for large amounts of data for reports on class assignments, and update grades, edit problem sets, or make comments on work. Here it is important to always have resources available for this kind of traffic in preparation for the next day, etc.

In more massively scaled systems, the presence of multiple time zones can greatly increase the period in which the types of traffic seen in the first model must be handled. While not a concern for our current goals, this would be a definite concern in any truly large scale system. Other considerations would have to be made for such a system which we will not address here.

2.3.3. Session Handling

A session is a record of the interaction of the system with a single user. Session handling is an integral part of any distributed web-based application. For scalability purposes there are two main possibilities. With *sticky* sessions, the load-balancer server will always forward requests for a given client's session to the same application server. If the sessions are not sticky, then each request within a session can be redirected to a different application server. In the sticky case the balancer needs to handle the incoming requests with a built in algorithm to balance the load equally among the application servers. One of the main downfalls of this solution is the lack of robustness - if one of the application servers goes down, then requests will be forwarded to a new server with a different session and e.g. you have to login again. With normal sessions, the sessions information will be distributed among the application servers and you can prevent the previous problem. One problem that arises here is in sessions validation, because the current session needs to be valid on all the application servers.

Assistment takes the non-sticky approach to sessions, for a few reasons. First, to avoid the problem mentioned above of losing session information in the event of an application failure. Losing connectivity during a tutoring activity due to lost session information is not desirable and is easy to avoid by handling sessions in this way. Secondly, to have a validation problem with the non-sticky sessions, a user would have to direct their browser at a particular machine in the cluster, which is unlikely at best (since there is little chance for our users to know the addresses of individual machines).

2.3.4. Database

The back-end can be a potential scalability problem for the system. Using one copy of the database can be a serious single point of failure. If the database goes down the system will stop to handle requests and to store important information. In an ITS application that stores information on student progress, losing information like this is unacceptable. Using more than one database can easily prevent this problem. One performance bottleneck that occurs is the synchronization cost of the databases, which can happen both in realtime (by inserting into both databases at once) or later at a specific time. A nice benefit of the first solution is that it makes it possible to spread reads between two databases in a round-robin fashion and increase the performance of the reads.

The problem is with insert, update, and delete queries that we cannot spread, because we have to process them on both databases and they will consume more resources than reads. This can lead to a possible scalability problem. One logical step can be to apply different partitioning operators like horizontal or vertical partitioning which can spread the writes as well and apply some application logic to connect to the correct data group. An easier solution is to use a good caching solution like memcached [2]. With memcached one can reduce the database reads by a significant fraction and the database can focus on infrequent writes decreasing the data access latency of the system.

2.3.5. Caching Content

A common strategy in distributed, web-based applications is to cache frequently accessed content. This can greatly reduce the read load on the database and provide much quicker response times. In a large, class-based, tutoring system, caching strategies can be expecially effective, since many of the students in the class are likely accessing the same content at the same time. Caching the problem sets that are shown to the first student will greatly improve the response time to subsequent students.

In general, there are two different strategies for propagating changes to cached data into the back-end database, *write-through* and *write-back* (also called *write-behind*). In a write-through cache, all changes to cached objects are immediately written to the database. In a write-back cache, changes are not written to the database until the object is removed from the cache. A write-back cache is often good for saving a database from load caused by multiple writes, since a number of changes to the cached object are never seen by the database. In contrast, write-through caches avoid frequent reads to the database by guaranteeing the state of the object in the cache matches the state of the object in the database.

The nature of the content served by an ITS is such that a write-through cache often provides excellent performance. The objects that are chosen to cache, primarily problems, problem sets, and logs of problems and problem sets, are largely static after creation. Most database writes are on the creation of these objects, which would have to happen anyway in a write-back cache, so few database writes are avoided, while many reads are.

For reasons of robustness, having more than one cache on more than one machine may be desirable. If the only cache goes down, there will be a severe performance degradation as the database must suddenly handle *all* the read requests. To have acceptable behavior from multiple caches, however, there has to be a policy for synchronizing the content of the two caches. That is, every time an object is added to or removed from one cache, the same operation in synchronously performed on the other. This has the benefit of guaranteeing the state of the caches remains the same at the cost of some latency in performing the synchronization. The load between the two caches can be balanced by randomly choosing between the two, and if the connection to one cache is lost, the system can stop directing requests at that cache until it can be brought back online.

3. Results and Discussion

3.1. Data Collection

Before discussing data collection and analysis in depth, it is helpful to see some more detail about the machines in the system, shown in Table 1. The system is a heterogenous environment involving multiple platforms and operating systems. In the table (lb) indicates the load balancing server, which in our case is also an application server. The (db) indicates the main database server, and (ap) indicates a dedicated application server. The server as3 also runs the memcached web cache described in 2.3.5.

As part of the ongoing processes used to monitor our system, data is collected about the processor usage on each of these machines. The data is collected by taking a single sample of the processor load of each machine at 5 minute intervals. The data collection process runs on a machine external to the application cluster, so its operation does not affect the results. This gives a total of 288 sample points each day. Each of these sets of sample points is saved, so a log of the performance each day is preserved. This is part of a system monitoring process which provides live statistics about the performance of the servers throughout each day.

For the data shown here, we chose a sampling period of 29 days, including weekends, that avoided public school vacation weeks. A period of multiple weeks allowed us to average data across a number of week days to show what a typical school day looks like. In particular, for each hour, there were 12 data points per hour, times 21 week days over the 29 day period, for a total of 252 data points per hour.

3.2. Analysis of Results

Figure 2 shows the relationship between the load and the number of users over an entire month. The scale for the processor load is in percent CPU utilization, and the number of users was divided by 100 to show it on the same chart. An “active

Table 1. Architecture of Existing System

Server	Uptime	OS	64 Bit?	CPU	Total Memory	Swap
as1 (lb)	$> 200days$	FreeBSD 6.2	Y	Intel Xeon 3.00GHz	8G	$< 1\%$
as2 (ap)	$> 200days$	FreeBSD 6.2	Y	Intel Xeon 3.20GHz	4G	$< 1\%$
as3 (ap)	$> 200days$	SuSE Linux 9.1	Y	Intel Xeon 3.20GHz	4G	$< 1\%$
as4 (db)	$> 600days$	Debian 3.4.3	Y	Intel Xeon 3.20GHz	8G	$< 1\%$
as5 (ap)	$< 100days$	SuSE Linux 9.3	Y	Intel Xeon 3.20GHz	4G	$< 1\%$

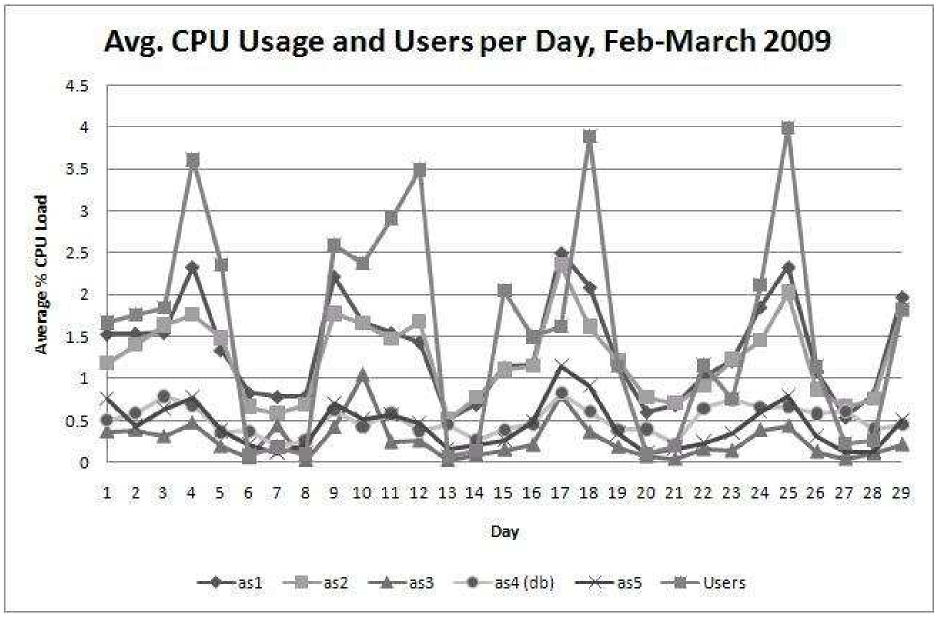

Figure 2. Average Users and CPU Load Over One Month

user" is defined here as a user who started at least one problem during a given day. Here the load of the application servers is closely related to the number of active users. Days with heavy server load clearly occur on days where a number of users were active, and this is true of all the application servers, which indicates that the load balancer divides the requests between the servers in a reasonable fashion. The differences between as1 and as2 and the rest of the servers is due to a combination of factors, of which the difference in operating sytems is likely important.

Here the difference between the two models of load is seen as well, on a large scale. It is clear that there are weekend days where both the number of users and the load on the application servers are low, which contrasts with weekdays which have many users and a correspondingly higher load. The low load corresponds with the second model, and the high load on weekdays with the first.

In Figure 3, values were averaged for particular hours of the day for all of the *weekdays* in the sampling period. This gives a picture of what the load looks like for a typical school day. Here an "active user" was a user who started at least one problem in a given hour. Here both models of load are also seen, in

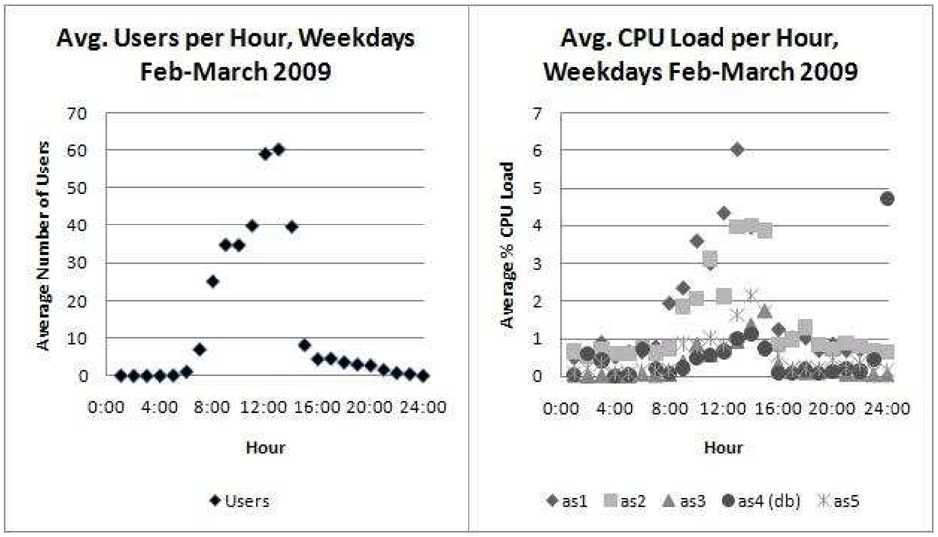

Figure 3. Average Users and CPU Load by Hour

more detail. From roughly 8 AM to 4 PM, the system is under heavy load as the number of students using the system is higher during the school day. When the data isn't averaged over multiple hours, it is also clear that the load rapidly changes as multiple classes start and stop. During the rest of the day, there is relatively little activity in the system, although in the evening there are at least some users on the system every hour. This group includes students completing homework, but in these charts does not count content creators or teachers, as they are not answering tutoring content

The hourly chart also shows that the system takes advantage of the low load at nighttime to run heavy database scripts for both the system and researchers, as seen in the activity on as4 (db) in the midnight hour.

While the hourly chart shows that on average there are 40-60 users during busy times, it should be noted that peak hours can be much higher than this. In the sampling period for this data, there were several hours where well over 100 distinct users were on the system, with the highest in the period measured being 192 users in an hour.

The load sharing which is shown here has important implications for scalibility. As long as the load is being balanced in such a fashion, in our case, more application servers can be added to scale the system up to a point. The number of users will then be limited by bottlenecks like the database server, the frontend load balancing server, and the single cache.

For a system like this, wher the performance is acceptable under load, the next major consideration is robustness. The current system is vulnerable to the failing of the same three components that represent performance choke points - the database server, load balancer, and cache. For scaling up, adding additional application servers will be important, but adding an additional cache, load balancer, and database will address both considerations of robustness and of scalibitily.

4. Conclusions and Future Work

The existing infrastructure of the existing system is more than adequate for the load that is currently handled. However, it will face difficulties with expansion when faced with bottlenecks including the load balancer, the single cache, and the database server. The proposed infrastructure in Figure 1 has the potential to support the entire state of Massachusetts. This is primarily because of the opportunity to expand the system in many areas, especially at the frontend at the load balancer and the backend at the database.

The proposed infrastructure offers not only an increase in the breadth of support, but also ensures that the system is robust with respect to 24/7 service to its users. In the case of a system expanding to support multiple areas of the country or the world, this is a major concern because of the different time zones involved. This would drive the system towards the first model of load more of the time, where a constant, heavier load is placed upon it.

Problems exist in extending the infrastructure moving forward. There need to be solutions for synchronizing the data in multiple caches, and in multiple databases, in an efficient and effective way. For example, one could use existing database techniques like horizontal and vertical partitioning, or de-normalization to scale up the backend. Applying one of these operators can help to reduce the query response time and achieve better performance. For cache synchronization, the system can take the benefit of a functioning web-cache [13] that can function as a balancer for multiple caches.

After the current iteration of scaling up is completed, these and other considerations will be considered by this project for the future. For any web-based tutor project with the goal of supporting large numbers of users, all of these scalibility and robustness issues are important to consider at the outset.

References

[1] Carp. *http://www.freebsd.org/doc/en/books/handbook/carp.html*, April 2009.

[2] memcached. *http://www.danga.com/memcached/*, April 2009.

[3] Pgpool-ii. *http://pgpool.projects.postgresql.org*, April 2009.

[4] J. Anderson, N. Kushmerick, and C. Lebiere. *Rules of the mind.* Lawrence Erlbaum Associates, 1993.

[5] P. Brusilovsky and C. Peylo. Adaptive and intelligent web-based educational systems. *Int. J. Artif. Intell. Ed.*, 13(2-4):159–172, 2003.

[6] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language.* O'Reilly Media, Inc., 2008.

[7] N. T. Heffernan, T. E. Turner, A. L. N. Lourenco, M. A. Macasek, G. Nuzzo-Jones, and K. R. Koedinger. "The ASSISTment Builder: Towards an Analysis of Cost Effectiveness of ITS creation". In *FLAIRS*, Florida, USA, 2006.

[8] R. M. Lerner. Open-source databases, part ii: Postgresql. *Linux J.*, 2007(157):16, 2007.

[9] S. Ritter and K. R. Koedinger. An architecture for plug-in tutor agents. *J. Artif. Intell. Educ.*, 7(3-4):315–347, 1996.

[10] Z. A. Shaw. Mongrel server. *http://mongrel.rubyforge.org*, April 2009.

[11] I. Sysoev. nginx. *http://nginx.net*, April 2009.

[12] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running.* O'Reilly Media, Inc., 2006.

[13] D. Wessels. *Squid: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.