WPI-CS-TR-08-02 January 24, 2008

Lessons Learned from Scaling Up a Web-Based Intelligent Tutoring System

by

Jozsef Patvarczki
Shane F. Almeida
Joseph E. Beck
Neil T. Heffernan

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Lessons Learned from Scaling Up a Web-Based Intelligent Tutoring System

Jozsef Patvarczki, Shane F. Almeida, Joseph E. Beck, and Neil T. Heffernan

Worcester Polytechnic Institute
Department of Computer Science
100 Institute Road
Worcester, MA 01609
{patvarcz,almeida,josephbeck,nth}@wpi.edu

Abstract. Client-based intelligent tutoring systems present challenges for content distribution, software updates, and research activity. With server-based intelligent tutoring systems, it is possible to easily distribute new and updated content, deploy new features and bug fixes, and allow researchers to more easily perform randomized, controlled studies with minimal client-side changes. Building a scalable system architecture that provides reliable service to students, teachers, and researchers is a challenge for server-based intelligent tutors. Our research team has built Assistment, a web-based tutor used by hundreds of students every day in the Worcester and Pittsburgh areas. Scaling up a server-based intelligent tutoring system requires a particular focus on speed and reliability from the software and system developers. This paper discusses the evolution of our architecture and how it has reduced the cost of authoring ITS and improved the performance and reliability of the system.

Key words: ITS, web-based tutor, scaling, architecture

1 Introduction

The idea of Web-based intelligent tutoring systems has been around for many years. Ritter and Koedinger suggested the possibility transitioning to Web-based tutors as early as 1996 [3] and he predicted many of the performance problems that we experience nowadays.

Server-based tutoring systems have many advantages over client-based approaches [1]. Accessibility is an important concern for tutoring systems. Students, teachers, and content creators all must have access to the system. Because of widespread Internet access, Web-based tutoring systems have the potential to provide access to many more users than can be reached with client-based software. Brusilovsky has claimed that Web-based systems have longer lifespans and better visibility than client-based [2]. Web-based systems virtually eliminate much of time and cost of installing software on individual client machines. Another advantage of server-based architectures is greater control over content distribution. Software updates and configuration changes are easily manageable with server-based architectures. Data collection is simplified by a centralized system. In addition, the data are available immediately in the form of reports.

The disadvantage of server-based systems is scalability as centralization of resources can create bottlenecks. This paper describes our ongoing efforts and solutions for addressing scaling problems in our system. The first part of the paper discusses different architecture problems and solutions through our research project evolution. Second, we propose a new hybrid architecture that addresses many scalability issues for ITS researchers.

2 Architectures

We were interested in architecture models for load-balancing and for fault-tolerance for online learning, which is becoming increasingly important [5]. In the case of a 7 days/24 hours service the reliability has great importance. The system needs to be active and responsible otherwise we can lose valuable students, teachers, and schools who have no time waiting for the service.

2.1 Comparisons

We tried to analyze different systems to get information about the common used infrastructure and solutions [4]. Furthermore, we compared our infrastructure with the existing ones. Our first generation system was a single application server connected to a single database without any monitoring environment. This caused our main problem the single-points-of-failure. When the application server went down our entire system was not available. We designed our architecture where the application servers appear to the users as one virtual computing resource. One approach we used was a virtualization by using Tomcat cluster with a master and a slave load-balancer and a virtual interface that makes the decision transferring the incoming requests for the particular application server.

We had symmetric clustered infrastructure where all the cluster members perform useful work. If one server fails the remaining ones continue the work. We had asymmetric load-balancer cluster where the backup balancer will take over the main server functionality if one event of failure will happen. Table 1 presents the compared systems and the description of the architectures. We used Apache Web-servers with mod_jk for load balancing purpose. Moreover, we applied Apache Tomcat connectorfor the correct communication between the cluster members to distribute the valid sessions. Our load-balancer algorithm was based on round-robin selection to distribute the load between the cluster members.

If we would like to compare the architecture of the Assistment system to the other ones we can realize that our system uses common symmetric clusterized application servers with an additional asymmetric load balancer farm and a virtual protocol that defines the virtual IP and does the failover switch. Our system has a well defined virtual IP that can hide all the other component of the system and gives security against the possible incoming attacks.

Table 1. Compared systems and the description of the architectures

System	Scalability	Database
WebCT	It supports improved scalability and load balancing through multiple servers.	Multiple databases, file server files stored on different file systems. No specified database type.
Wikipedia and Wikimedia	Several racks of full servers, Apache web servers, Squid systems for large cache of pages, round-robin load-balancing.	Master-slave database architecture (MySQL), and search servers.
MySpace	Totally distributed computing architecture, physically separated computers that can form a logical one.	Single database for user details, multiple database servers for data, load-balancing among database servers (Microsoft SQL Server), two copies of SQL Server on each server computer.
eBay	The architecture is a type of Grid computing that allows for both error correction and growth. Apache Web-servers, application servers, search servers, data servers, and proxy servers that are balanced using a random selection algorithm. Sticky load-balancing.	Couple hundred of database servers and search servers. Load-balanced database infrastructure. No specified database type.
Assistment	Virtual interface, asymmetric load-balancer cluster, symmetric application servers, load balancing with round-robin selection algorithm. Hybrid client-server architecture.	Prototype fault-tolerance master-slave cluster with synchronized databases (PostgreSQL).

2.2 First Generation — Tomcat Implementation

Based on the analysis and the answers we decided to analyze our system at more deep level. First, we were interested in the communication between the application server and the database server. We set up a test environment with one application and a database server that used a browser to play student and tried to capture the time and the number of reads and writes at the back-end. By the help of this analysis we were able to detect the problematic actions in time.

Based on the result we realized that the "Start Work" and the "Resume Work" actions generated the most reads which can be a bottleneck of the system, as shown in Figure 1.

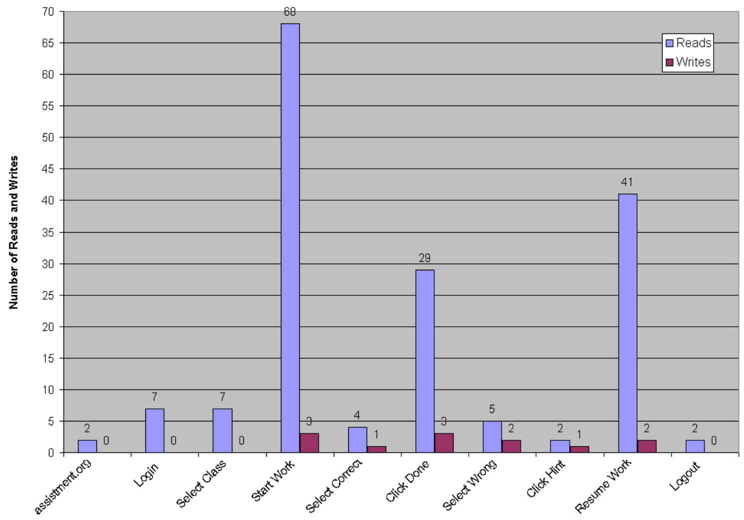

Fig. 1. Number of reads and writes for each action

The incorrect infrastructure can affect the productivity and reduce the up-time of the entire system. The first version of the system was in Java. Our infrastructure contained two Web-servers, two Tomcat application servers, two load-balancers (master-slave), Web-cache system, and one back-end database server. The architectures of the first generation Assistment system is represented by Figure 2. We had asymmetric load-balancer server environment with symmetric Tomcat cluster where we used sticky session handling to claim always the same application server for the user after the first request.

Additionally to the common balancer infrastructure we designed and applied the Common Address Redundancy Protocol (CARP) protocol at the top of the balancers for preventing the system from the single-point-of-failure. In the case of any problem with the master balancer, this protocol ensured the automatic

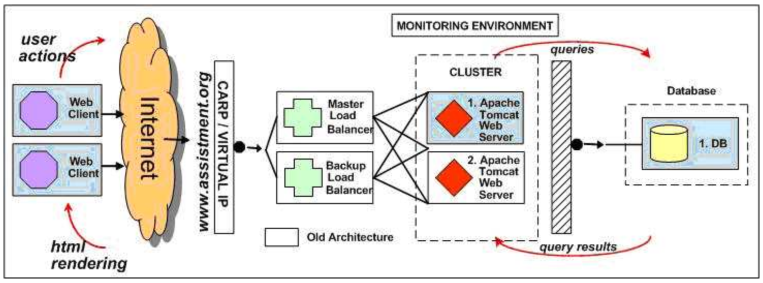

Fig. 2. Architecture of the first generation Assistment system

switch between the master and the slave balancer. At the front-end we had a virtual IP address handled by CARP. Behind this IP the actual balancer was able to find by the incoming requests.

Our software has four main components: 1) a "portal" though which students and teacher log into, 2) a "runtime" where students get tutored, 3) a "reporting" system where teachers get reports on what their students have done, and 4) a "content builder" used by teachers and researchers at CMU and WPI.

Given that we run about 100 classes, we wish to see how far we are from supporting all the 8th grade students in the state of Massachusetts. There is no direct data about how many simultaneous users we would have to support for the entire state, though we can provide a back-of-the-envelope calculation. We have estimated that 75,000 users could support the entire state of Massachusetts. In an ideal situation, these students would use our system once every two weeks. Therefore we would need to support around 7500 students a day. Given that there are on average 7 class periods in a given day, our next generation system would have to support about 1100 simultaneous students.

We have determined that a significant portion of time in our runtime environment is spent retrieving student content and creating a java object that represents this, through what we called our Data-layer. This content was serialized as Extensible Markup Language (XML) and stored in a relational database. We noticed that during this retrieval our bottleneck was retrieving and parsing this XML. We put in place a cache that stores this parsed XML, with replacement policies for when content is changed. This not only removed this bottleneck, but also increased the performance of this subsystem by 8 fold. We have estimated that this will increase our systems capacity by a factor of at least 2. The server applications are more difficult to analyze and debug than the standard stand-alone ones.

With the old system and infrastructure the most important result shown in Figure 3 is that all but a handful of these 10-minute interval have average page creation times that are less then 1 second. Most of these are clustered around 100 milliseconds. Since each public school classes have about 20 students, we noticed clusters (shown in ovals in the bottom left) of intervals where a single class was logged on. We noticed a second cluster of around 40 users, which most likely represents instances where two classes of students were using the system simultaneously. Surprising to us, even the intervals where 80-100 users were on the system simultaneously, there was no appreciable pattern towards a slower page creation time. This was true, even when we had 80-100 users one the system and we were generating over 1000 pages per interval (100 pages per minute). Of course, with more users, we expect to see a slow down, but we have not yet reached that point. Additionally, there are a few intervals with an average page creation time greater then 1 second (top-left oval). We inspected our logs, and found out those intervals were associated with times when an entire class of students would log on simultaneously. The log-on procedures is the most expensive step in the process and this data shows that this might be a good place for us to improve.

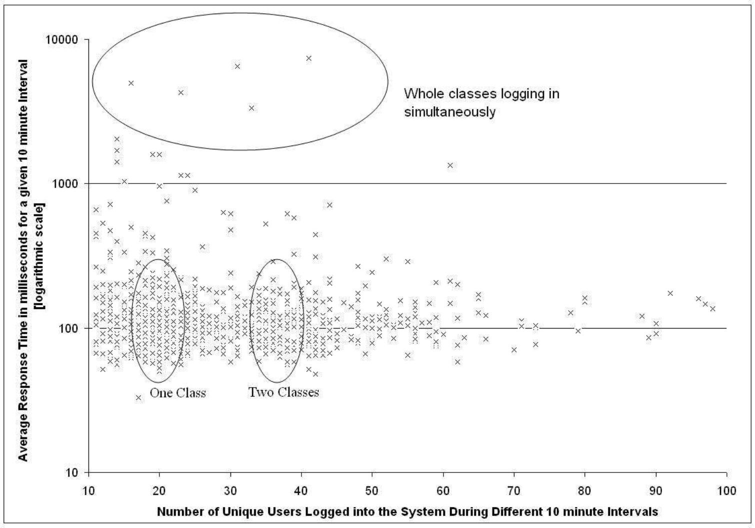

Fig. 3. Average response time of the first generation Assistment system

2.3 Second Generation — Tomcat Implementation with Scalability

We wanted to evaluate our decision to increase scalability by adding an application server to our load-balanced infrastructure. Our research question was whether we could get a linear speed-up with additional Tomcat servers or not. We also wanted to see if a caching method could increase performance. In our methodology we proposed to measure the speed up by simulating students using the Apache JMeter framework. We simulated three different scenarios. In the first scenario we used 50 threads simulating 50 students working with our old infrastructure (no load-balancer, one tomcat box, and one database server). We used an average 10 seconds random delay between each student action that represents a typical student reaction time. In our second scenario we changed the infrastructure and we used the load-balancer to distribute the load between two servers (load balancer, two Tomcat boxes, and one database server). Finally in our third scenario, we used a Web-cache technique without a load balancer and one Tomcat box with a single database. During the execution of the tests the memory and CPU utilizations were normal at the application server and at the database side as well. For the test, we used a Pentium 4 with 3GHz CPU and 2GB RAM. Table 2 represents the results for the test scenarios.

Table 2. Test results showing the load balancer cut the average response time in half

Test Type	Number of Unique Users	Response Time (ms)
Scenario 1/Normal	50	8955
Scenario 2/Load-Balancer	50	3624
Scenario 3/Web-cache	50	8073

According to the results of the first scenario we got 9 seconds for average response time, which looked to high. During the test we "played student" and found that the average response time was less than 9 seconds, which led us to conclude that JMeter added a small latency. Nevertheless, it does seem that the load balancer did appear to cut the response time in half; however, the caching method had a much less significant role in reducing the average response time. We seem to have able to get linear speed-up by the help of the load-balancer and an additional Tomcat server.

2.4 Third Generation — Reimplementation in Ruby on Rails

As our project progressed, the complexity of our Java source code reached the limits of maintainability. The size of the code base, contributions by many different developers, and thousands of lines of uncommented code made the system difficult to learn. We also wanted to eliminate the XML objects to get more speed-up. Moreover, we designed an expandable system with a flexible user interface that is simple to learn and use from the first moment, and it is easy to expand and add new features or modules. For us it was important to keep reducing the response time of each request and actions. Moreover, we wanted to change from Java to different programming language because of the inconsistent developed code-parts and to speed-up the development process.

At the same time, a separate effort was underway to revamp the user interface. Ruby on Rails, a free web application framework based on the open-source Ruby programming language, was used to create mockups of new interface components. The framework support provided by Rails and the simplified syntax of Ruby allowed for rapid prototyping of the interface. The ease with which the new interface was constructed using Rails convinced the development team to use Rails for the reimplementation of the system. The nature of Ruby allowed us to rewrite the core system in weeks rather than months. In the process, we reduced the lines of code from roughly 20,000 to around 5,000. The significant reduction in lines of code greatly improved the understandability of the code base.

Because Rails and Java are fundamentally different systems, our infrastructure required a redesign to accommodate the new system. Mongrel, a single-threaded web server for Ruby applications, is used to serve content. Because it is single-threaded, multiple Mongrel application servers are used concurrently in a cluster. A simple Apache Web server provides load balancing to the cluster and a single PostgreSQL database server stores all content and data. Since being deployed in September 2007, the system has been used by approximately 5000 students and hundreds of teachers.

As more students began using the system simultaneously, we encountered severe scaling problems. Examining loads on the application and database servers revealed that the database was a serious bottleneck in the system. During peak times, the database server was constantly at maximum capacity while the application servers remained nearly idle waiting for data. Because of the slow database response time, the incoming request rate quickly exceeded our service rate. Examining the most expensive operations in the system revealed some implementation flaws in our system, particularly in our use of the database abstraction features of Rails. Although these design problems affected both students and teachers, the affect on students was especially troubling.

According to log data, the tutoring portion of our system spends an average of 2.27 seconds processing a request (standard deviation of 3.39). With six Mongrel servers, our system can only handle approximately three requests per second. Indeed, according to the same logs, over 90% of tutor actions during school hours occurred when the request-per-second rate was three or less. An obvious remedy is to add additional Mongrel servers to boost our maximum throughput. Unfortunately, adding just three more servers to the cluster overwhelmed the database and did not result in increased performance.

We calculated the average response time of the most expensive operations in the tutor. Figure 4 presents the average response time of the common actions including the most expensive ones: account creation ("Signup"), the welcome page ("Account"), class assignment page ("Assignment"), and loading the first problem of an assignment ("First Problem"). We captured the average response time of the created test curriculum ("Average Response Time") and the total number of users for that particular day (indicated above the each cluster). Each cluster represents a different day, beginning on a Saturday during a three-day weekend. The graph shows that response time in each cluster appears to be independent of total users. This suggests that our scalability problems begin at very low usage.

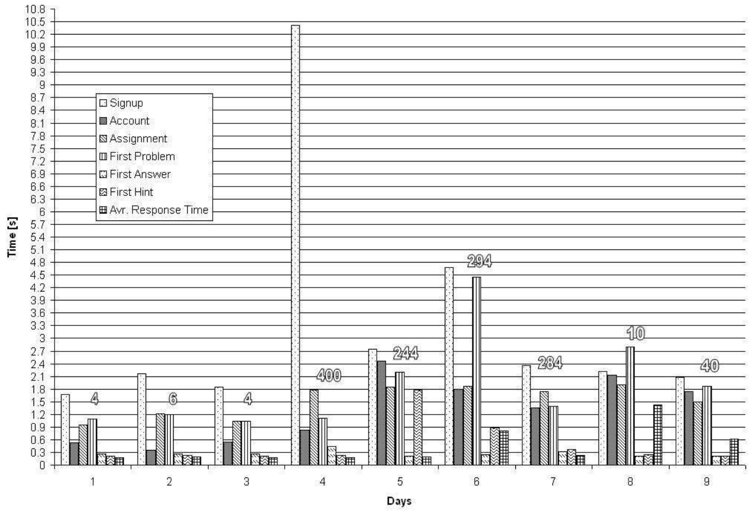

Fig. 4. Average response time of the third generation Assistment system

2.5 Fourth Generation — Rails with Thicker Client Architecture

The revelation of design problems prompted a shift in our thinking. Although we could have conceivably fixed the system so that it could handle typical loads in a timelier manner, we decided to focus on more fundamental changes that would allow the system to scale well beyond our current user base. Instead of handling all data processing ourselves, we decided to take advantage of the computing power of our users by pushing many of the basic tasks to the clients. However, we still wished to avoid a thick-client implementation for many reasons like in the case of the Push and Pull approach of CUMULATE and CUMULATE2 [6].

One of the most important reasons for retaining a thin-client model is accessibility. Beyond a reasonably modern web browser with support for JavaScript, no special software or third-party applications are required to use our system. The lightweight interface means our software can be used in schools with limited budgets for computing resources. In addition, having control of content distribution and software updates reduces some of our maintenance burden. With a server-based architecture, all changes in content and software happen on our systems and we do not need to push updates to clients. This virtually eliminates the need for in-field debugging and maintenance. Finally, a server-based system gives us greater flexibility in terms of research activity. As with content distribution and software updates, we have greater control of experiments by running them from our servers.

For the reasons described, our servers are still responsible for on-demand content distribution and log gathering and retention, while data processing (e.g., determining the correctness of answers) has been re-implemented in JavaScript and is now the responsibility of the client system. This redesign has reduced load on the database dramatically. At the start of an assignment, the client requests and stores questions asynchronously while the student begins working. Network or server delays are hidden by the asynchronous communication. In addition, because the content is stored on the client, the progression through an assignment is handled by the client with only log messages sent back to the server. In our previous Rails implementation, the database server was overwhelmed by the volume of requests from the clients for relevant in-tutor content (e.g., hints). That content is now packaged together in advanced and sent in a single message to the client. When a student requests help, his browser simply has to render information it already has instead of sending a request to our servers.

Latency is a frequent problem for Web users. Stern, Woolf, and Kurose discussed the idea of prefetching content in their MANIC system [4]. In initial testing of the JavaScript implementation of the tutor, our system spends an average of 0.08 seconds processing a request (standard deviation of 1.43). With just six Mongrel servers, our system is now posed to handle 75 requests per second. Even if our throughput does not match this estimate, latency hiding via asynchronous requests will help provide a fast user experience.

3 Future Work

Our newly developed system focuses on the flexibility of the ITS, the implementation of the system principles, the code portability and productivity, expandability, and the client-server hybrid solution. However, the database is a single-point-of-failure and a significant bottleneck in the system. The architecture has to include a fault-tolerant back-end. We currently have Assistment content hosted as a Web service. Because of this, by implementing a Web service interface, another ITS could incorporate Assistment content into their system. The Assistment project is looking to collaborate with other systems to include various features. This is just one example of how web services have enabled the Assistment project to become more collaborative and extensible. Intelligent Tutoring Systems are customarily thought of as closed systems. With such a strong research community, it would seem beneficial for researchers to be able to work together; using contents and tutoring methods from within other existing system. With this collaboration, existing tutors could reach broader audience.

Our goal is to provide reliable and effective tutoring to hundreds of thousands of users. Although we do not have a total solution yet, we are actively exploring architecture designs that will allow us to meet the goal.

4 Conclusions

We have reported some positive results with regard to scaling up ITS. Clearly, many of the issues discussed in this paper are specific to our implementation. However, the underlying architecture decisions we have made are of general interest to the Web-based tutoring community. Server-based systems allow greater control of content distribution, system configuration, and software maintenance than client-based systems. In addition, server-based systems, particularly Web-based systems, offer greater accessibility to users. Finally, a server-based system has the benefit of immediate data collection: teachers and researchers have access to the same information in real-time. Unfortunately, server-based systems face scalability problems. The hybrid design we are currently exploring retains many of the advantages of server-based systems while addressing these scalability concerns. The evolution of our system and the result of this research is useful for the entire ITS community. Sharing the results of our work with other ITS researchers will help expand the development of server-based tutoring systems.

5 Acknowledgements

We would like to acknowledge the Department of Education, National Science Foundation, Office of Naval Research, and the Worcester Public School System for providing the means to accomplish this research.

References

1. B. Bacic. Constructing intelligent tutoring systems: design guidelines. *Information Technology Interfaces, 2002. ITI 2002. Proceedings of the 24th International Conference on Information Technology Interfaces*, pages 129–134 vol.1, 2002.
2. Peter Brusilovsky and Christoph Peylo. Adaptive and intelligent web-based educational systems. *J. Artif. Intell. Educ.*, 13(2-4):159–172, 2003.
3. Steven Ritter and Kenneth R. Koedinger. An architecture for plug-in tutor agents. *J. Artif. Intell. Educ.*, 7(3-4):315–347, 1996.
4. M. Stern, B. Woolf, and J. Kurose. Intelligence on the web? In *8th World Conference of the AIED Society*, 1997.
5. D. Suthers. Representational support for collaborative inquiry. *System Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, page 14 pp., 1999.
6. Michael Yudelson, Peter Brusilovsky, and Vladimir Zadorozhny. *A User Modeling Server for Contemporary Adaptive Hypermedia: An Evaluation of the Push Approach to Evidence Propagation*, volume 4511/2007 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, Connecticut, USA, 2007.