10,001 (Or More) Polling Stations:

Voting Virtually @ UAF

Joshua J. Kugler

jkugler@bigfoot.com

Version 1.2
Revised: 2004-07-07

Abstract

One of the major struggles at the Associated Students of the University of Alaska Fairbanks is the fight against student apathy; probably ranking in the top five priorities for the organization. Anything which would precipitate more involvement by students, whether directly in student government, or indirectly in other campus activities, is encouraged and promoted. One way in which ASUAF decided to fight student apathy, and save money, was through the use of online elections. A higher turnout was hoped for, and realized.

In the process of moving to online elections, many design issues and considerations had to be collected and dealt with. Security, authentication, confidentiality, anonymity, and stability were paramount in the process. Stepping through each of these areas carefully lead to virtual voting at UAF, and a successful online election.

10,001 (Or More) Polling Stations:

Voting Virtually @ UAF

Introduction

I had been working for the Associated Students of the University of Alaska Fairbanks (ASUAF) for two and a half years. During that time, one of the issues high on the list of priorities had been conquering student apathy. This apathy had been evidenced in the low voter turnout at ASUAF elections, the lack of broad-based student involvement in student government, and a general lack of concern for issues being decided upon by the ASUAF Senate, unless, of course, it directly affected a club or cause in which the student was involved.

In the fall of 2002, an idea for reducing student apathy, and increasing student involvement, was floated. Several justifications were used to promote this idea. First of all, it would save money: there would be no need to print paper ballots, nor would manned polling stations be mandatory, thus reducing the amount paid to poll workers. The biggest benefit, though, would be the hoped-for higher voter turnout. In previous elections, polling stations were situated in four locations: the Downtown Center (a UAF extension in downtown Fairbanks), the Lola Tilly Commons (cafeteria), Wood Center (a student union of sorts), and Arctic Health (a building on the "other end" of campus, away from the main campus). Even with these locations, voter turnout was still low. This was especially true at the Downtown Center, as many of the students there took night classes, or had sporadic schedules, which caused them to miss the voting booth hours.

This idea was online voting.

Design

Many decisions had to be made once the official approval was given to Online Voting. These included: which platform(s) we would use; the server environment under which the voting application would run; requirements for structure, security, and anonymity; as well as issues which did not have to be dealt with before electronic balloting, such as authentication, needed modifications to the election manual, how to handle question ballots, and how to ensure we had true anonymous balloting.

Timeline

We began discussing online voting in the middle of October, 2002, with the official green light on the project being given about the first of November. Our election was to take place December 3 and 4 of that year, a little over a month away. This made for a very tight schedule: requirements gathering, design, implementation, and testing would have to take place in four weeks. The list of tasks was long, the time was not.

Platform

Operating System: Linux

The operating system platform chosen for this project was Linux. The reasons for selection included its excellent security record, low operating cost, high ROI, rock-solid stability (at that time, our web server had an uptime of over 330 days, and made it past 550 before a power outage forced a shutdown) and Open Source nature. The kernel installed, as of the execution of the project, was a custom kernel, version 2.4.17, compiled from clean, unpatched sources obtained from kernel.org. The distribution used was Mandrake Linux 8.2. All packages below, however, were custom compiled using sources obtained from their respective web sites, and not installed using Mandrake RPM packages.

Web Server: Apache

The web server used was Apache 1.3.27, static build (no Dynamic Shared Objects). It was chosen, like Linux, due to its cost, speed, stability, and excellent security record. In addition, factors such as third-party module support (see mod_ssl below), and excellent compliance with standards played into the decision.

Database Server: MySQL

Our database of choice was MySQL Server 3.23.51. In a project like this, speed is paramount. Votes must be recorded, and a session closed, as quickly as possible. MySQL delivers, with flying colors. In a recent test by eWeek, MySQL 4.0.1 (the successor to the version we used) tied Oracle 9i for first in throughput and response times (Dyck, 2002). Its low cost, as well as its excellent interoperability with the Perl programming language, also played into the decision. There has been support for transactions recently added as well, making it possible to easily roll-back insertions of voting data, should an error or aborted connection occur. MySQL also has very fine-grained security controls, which are discussed in more detail in the *Safeguards* section below.

Programming Language: Perl

For the server-side implementation of the voting system, the Perl programming language was used. Perl, or Practical Extraction and Report Language, was used for a number of reasons. Because it is a script-based, interpreted/compiled[1] language, there is no need to compile sources every time a change is made to the source code. This allows for rapid prototyping and "low cost" experimenting: there is no need for edit/compile/test cycles. Simply edit and run. Perl's untyped variables also make it a natural for processing input from web-based forms, as there is no need to know before hand what kind of data will be in a form. This is especially useful when processing the results of dynamically generated forms, such as are used in our voting system. For example, in doing surveys, we never know (until the form is generated) whether we will be receiving numerical, check-box, type responses, or a free-form text field. Perl also has strong support for built-in hashes and safe arrays. These also come in handy for processing web input.

Security: OpenSSL and mod_ssl

In a project such as this, the security of data transmission is always paramount. For encryption of data en-route, we chose OpenSSL for the libraries, and mod_ssl for the web server interface to those libraries. OpenSSL "is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library [and is] based on the excellent SSLeay library developed by Eric A. Young and Tim J. Hudson. (OpenSSL Team, 2002)." Mod_ssl is a module which "provides strong cryptography for the Apache 1.3 web server via the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols by the help of the Open Source SSL/TLS toolkit OpenSSL. (mod_ssl Team, 2002)." OpenSSL and mod_ssl were chosen because of their low

cost, open source nature, and excellent security record. In addition , mod_ssl was chosen because of its quick and seamless integration with the Apache Web Server. The time required to download, configure, and compile mod_ssl into an already working Apache server was less than an hour.

Environment

In the design process, a decision with regards to the environment under which the voting system would run had to be made. We had two options: suEXEC, which was already installed, or a dedicated server. We chose to go with suEXEC for two reasons. 1) It saved the time and money of installing a new server, which was a very real concern as we had less than a month from project authorization to voting day; 2) It provides many of the benefits of a dedicated server without the logistical and support problems a second server would bring. We discuss this more in the *Safeguards* section below.

Requirements and Issues

Authentication and Identity Verification

One of the foremost concerns for any online voting system is that of authentication and identity verification. It is imperative that there be a mechanism by which we can guarantee, with reasonable certainty, that the person at the other end of the data path is who they are claiming to be. We came up with a few ideas while we were trying to determine how to authenticate users.

One of those ideas was to use Aurora, our campus-wide e-mail server. Every student at UAF has (or should have) an e-mail account on Aurora, thus has a valid user ID and password, which is exactly what we are looking for. This would not work, however, because there was no sure way of determining voter eligibility; there are accounts with expired passwords due to non-use (unfortunately very common); and some accounts no longer had the real names in /etc/passwd, thus there was no way to determine the real user on a system where usernames are assigned by initials. But in addition to all these, there is no way to remotely authenticate from Aurora: no LDAP[2] and no NIS (YP)[3].

A second option was to try to use Banner, and its sister system, UA Online. Banner is our university-wide information system. It stores *everything* about the university system, including student records. It, in turn, is used by the UA Online system, a web site where students can check and pay balances, register for classes, and perform other functions that relate to their college career at UAF. The problem with using this system, ironically enough, was security.

Being the central store of university information, the walls around Banner, as you can imagine, are pretty high and pretty tight. Getting access to this system, even if it was read-only access to a few tables, would not have been worth the effort, and would not have been accomplished in the needed time frame.

The central issue in authentication is determining how to know that the user is who they claim to be. It must use something which only the user *has* or *knows*. Biometrics are "perfect," in a manner of speaking, but the goal of an online election is user convenience, and there are very few who own the needed hardware to implement such a system. Account names and passwords can be, and often are, shared. This brings us down to using personal information that only the user *should* know, and using multiple pieces to increase our certainty.

The solution we settled upon was that of using our campus PolarExpress system. PolarExpress is the name given to our proprietary AT&T card-swipe authentication system. This system is used for tracking meal plans, providing access to buildings, and keeping track of how much money is left in student accounts for copy machine use, among other things. Included in this database are records containing student name, university ID (usually social security number, but may be a number assigned by the university if the student is international, or otherwise doesn't wish to give their SSN), library card number, birth date, and number of credits in which they are currently enrolled.

To use this database, we obtained an extract, which includes the above information, of all students who were eligible voters (taking three credits or more). We then imported this into our database, and authenticated users by asking them for their student ID number and birth date, which were required. We also asked for their library card number *or* the number of credits in which they were currently enrolled. Having obtained this information, we compared this input against that stored in the database, and if we had a match—the first two fields match, and one of the last two—then we considered the user authenticated.

Anonymity

In any voting system, whether on paper, or online, anonymity must be preserved. There cannot be any way of tracing a particular vote back to the individual who placed it. This means that the relational integrity of the database must, in a way, be intentionally compromised: there will be two sets of tables, between which there is no primary/foreign key connection or constraints[4]. Each vote must be recorded in the database as a simple "tally mark," not linked in any way to the user record or session record from which it originated.

One Person, One Vote

In addition to guaranteeing the anonymity of the voters, at the same time there must be safeguards in place to prevent multiple votes by the same individual. In reality, this is rather simple. As a part of the login system, a record is kept of which voting sessions are currently open, and which are closed. If a user is logged in with an open session, and then tries to login again, an access denied message, along with an explanation, is generated. If their voting session is closed, which means their votes have been recorded, their login is likewise denied. If something happens during the voting process and their browser dies, denying them access to their voting session, they may wait a while, and after a set period of time with no activity, their session record will be deleted, allowing them to log in again.

Write-in Votes

In addition to the individuals on the official ballot, write-in votes must be accommodated. If a voter wants to vote for a candidate that is not on the ballot, there must be empty boxes on the form to enter the name(s) of desired candidate(s). These must inserted in a separate table: one for each time the name is put on a ballot.

When a voter submits write-in votes, there is the possibility of a name being written in multiple times. To try to catch this, all the write-in slots could be compared to make sure the voter did not write the same name twice or more. However, if we must allow for slight misspellings when counting ballots, then comparisons would not catch a voter writing in the same candidate using a slightly varied spelling. For example: they could vote for "Joshua Kugler" and "Joshua Koogler," and Joshua Kugler suddenly has two votes from one voter. We could do a SOUNDEX (or similar) comparison, which is an algorithm that allows us to compare the phonetic sound of two words (e.g. "their" and "there" should have the same SOUNDEX signature), but it is not perfect. The solution at which we finally arrived is discussed below, in *Recounts.*

Recounts

In a paper ballot system, if a recount is called for, all the ballots must be looked at again, and all the votes tallied. In an online system, it is (theoretically) as easy as running the report application again. But it might not be that easy. There may be disputes as to the validity of the votes, the system, or the counting mechanism used. The only solution: print out all the votes for the given election, and count them by hand.

To facilitate a true recount, it must be possible to reassemble all ballots, and in the process still guarantee anonymity. This was accomplished by using a ballot ticket. The first idea was to generate a ballot ticket using a table that would do nothing but generate a sequence. It was pointed out, however, that it might be possible, using the order in which people voted (which *is* tracked, by nature of the table structure), to approximate the votes cast by a particular user, since the ticket would be increased as each person voted. The order may be thrown off some since tickets would be generated when surveys are taken as well, but it still presents a real danger for vote correlation. Instead, we used a random ticket. The two tables used for recording votes (`ev_votes` and `ev_fitba`) have a six character alphanumeric field called `ticket_id`. Using upper and lower case letters, as well as the ten digits, this allows for 62^6 or $5.68*10^{10}$ possibilities. With that many unique possibilities, getting conflicting ticket ID's is not likely (but we check, just in case). When we start recording the votes for a particular ballot, we generate a unique `ticket_id`, and insert it along with every vote on that ballot. Finding these distinct values, and then printing out all votes using each unique value will allow us to reconstruct a ballot.

Modifying Paradigms

In moving toward online elections, we realized there might be a conflict with established rules in our elections manual. It would have to be modified to accommodate online elections.

1) The manual was very specific as to the location and number of polling stations. With the elections being online, a polling station could be anywhere. In the new manual, the existing polling stations are kept, but with kiosks for voting, instead of the paper ballots.
2) The manual also spelled out the format and design of the paper ballots that were used. The language had to be changed to allow for the ballot being displayed on the screen. One interesting thing in the old manual was the stipulation that the names on the ballot be printed in at least two different orders, as to not favor any candidates. This is accomplished quite easily with the electronic ballot by adding `ORDER BY RAND()` to the SQL query that retrieves the candidates. We can order the ballot, literally, a different way every time. It is very unlikely, in fact, that the ballot will be displayed the same way twice in the course of an election: with a ballot of 14 candidates, this presents us 87,178,291,200 ways to display the ballot.
3) Electioneering had never been officially illegal, but it was heavily frowned upon by ASUAF. With no real way to enforce electioneering now—how can a candidate prove there was undue influence imposed upon a voter or voters?—the concept of electioneering was completely done away with.

4) Another issue we had to deal with was that of what to do with question ballots. In the old system, when a voter did not have proof of their student status, they could show a picture ID, sign an envelope, place their ballot in the envelope, and place the envelope in the ballot box. When their eligibility was proved (by manual verification), their ballot would be counted with all the other regular ballots. The debate on how to translate this to the online system went on for some time. The idea was floated of letting a question ballot be used when the voter had less than some number of needed pieces of information for verification. Then, they could come back later when they had *all* their pieces of information, vote again, and then their question ballot would not count. The question ballot *would* count if they never came back and submitted a full vote. However, there were too many issues with the fact that a voter could be impersonated with enough information, and the vote would still count if the real user never voted. There was also the issue of having to keep track of votes issued on question ballots so they could be removed later if needed: this would, in fact, remove anonymity. The conclusion: no question ballots. Either you would be able to log in and submit a full, regular ballot, or you wouldn't.

Safeguards

In addition to the overall security designed into the voting system, safeguards were put in place, both on the server side, and the human side.

On the server side, two major precautions were used. One of these was the security system built into the MySQL database server. MySQL allows very fine-grained access control, being able to define access rights, starting with the host from which a user can log in, all the way down to defining the tables, and even the columns, on which the user can perform operations. Creating a user that can only read and write to certain tables in the database prevents the voting system from being abused should some compromise take place.

A second safeguard on the server is the use of an Apache extension called suEXEC. suEXEC allows a script to be run as the real (not just effective) UID of the owner of the script. This facilitates two things. First, any local configuration files or libraries used by our voting system can be readable only by the one UID, in this case the ASUAF account that owns all the ASUAF web site files. Second, all the scripts can by set 700 (or rwx------) so only the owner can read and execute them. This further enhances security on a shared system: no one other than the owner will be able to view passwords or other important configuration information contained in the programs and their associated libraries.

Two different precautions were taken on the human sides of things: an NDA (Non-Disclosure Agreement) and a non-tamper agreement with penalty of perjury.

With the exclusion of question ballots, there would now be theoretically no way to correlate a ballot with a voter. Should any method arise to link voters and ballots, however, the Information Services Department agreed not to disclose the method or the name/vote combinations. In addition, the members of the Information Services Department agreed not to tamper with the voting database. In other words, no electronic ballot box stuffing.

Procedures

Login and Session Creation

When a voter requests the login screen, a new vote object is created. This vote object contains the utility functions and routines to perform compliance checks and interact with the voting database. In fact neither `login.cgi` or `ballot.cgi` do any direct database access: it is all through the `Vote.pm` library. After successful creation of the voting object, it first checks to make sure there is currently an open poll (the system currently only supports one open poll at a time). If the poll is not open, the voter is told such. If it is open, he is presented with a screen asking for his student ID (which will usually be his social security number), his date of birth, and his library card number *or* the number of credits in which he is currently enrolled. After clicking Login, session creation is attempted.

The form is first checked to make sure all the required fields have been filled in. Next we check to make sure the voter has not already voted. If the voter has already voted[5], he is told such and no session is created. If both of these are OK, we proceed and attempt to create a session. The student ID and date of birth must match, and either the number of credits or the library card number must match. If these match, and he does not have a currently open voting session, a session is created. This session has a 64 character session ID obtained from the environmental variable `$SSL_SESSION_ID`. This session ID uses upper case characters and numbers, giving 36^{64} or $4.0*10^{99}$ possibilities. The voter is then redirected (using an HTTP 302/Location header) to the balloting module.

Printing Out the Ballot

The redirect command consists of the name of the balloting module (`ballot.cgi`) followed by a forward slash and then the 64 character session ID. The session ID can be obtained from the environmental variable `$PATH_INFO`[6]. We first create a voting object, passing it the session

ID as the constructor's only parameter. We exit if we cannot create a voting object. We then check to make sure 1) the session is still open (meaning the voter has not voted or has not filled out the survey), and 2) the session has not timed out. If either of those fail, we inform the voter and exit.

We then check for the HTTP request method. If the request method is GET, this would mean the voter requested the file (as opposed to POST'ing to it via a form), either through being redirected to it upon login, or clicking the link printed after the ballot has been submitted. At this point, we print out the ballot. If the voter has not voted yet, we print out the candidate/referendum ballot, otherwise we print out the survey. We retrieve all the questions for the applicable section(s), and print them to the browser.

Taking Input From Ballot and Checking Input For Validity

When the voter pushes the "vote" button, the answers are gathered from the form. Check box and radio button values are pushed to an array of arrays, each element being the ballot ID and answer ID. The write-in values are pushed to another array of arrays, each element being ballot ID and the text of the answer.

Once the answers are gathered, they are passed to a validating routine. The routine checks 1) to make sure there are no more than the maximum number of answers for a question and 2) all the answers provided apply to the current election and ballot section (preventing voting twice using a still open survey session). If the checks fail, the voter is notified, and the ballot is printed again. If the checks succeed, we proceed to recording the votes.

Recording Votes

We first call `begin_record`, which begins a database transaction, generates a unique ballot ticket, updates the voters session `last_action`, and closes their voting session (if we're recording ballot answers) or closes their survey session (if we're recording survey answers). If there are any database errors, the transaction is rolled back, and an error is returned.

We then record the checkbox and radio button items. It is here that duplicate multiple-choice votes are handled. If a balloted (non write-in) candidate is being voted for more than once, then it means the form has been tampered with (possibly by saving locally and editing). In this case, the insert will return an error since there is a unique index on ticket, ballot ID and choice number. If this is the case, it is simply ignored (silent fail) so that it will not be apparent to the user that their subterfuge did not work. Any other error will cause a rollback and the user and

webmaster will be notified. We then record the write-in (as well as fill-in) questions, checking for errors.

Once all the data has been recorded successfully, we commit the transaction. A thank you message is then printed out. If the user has just voted, and there is a survey, we print out a link to the survey.

Analysis and Debrief

Summary

Overall, the election went very well. There were a few hitches due to incorrect information provided by PolarExpress, or possibly students not being aware that they had (or had not) in fact dropped a class. In one case, a voter was not in the database because he was faculty, but also a student, thus the extract from PolarExpress did not catch those kinds of records. Throughout the 48 hours of the election, there were very few fires to put out.

Voter Turnout

Voter turnout was increased, but still below what was desired. We had a total of 260 ballots cast and 179 surveys filled out. The number of ballots cast was approximately a 25 percent increase compared to our last fall election. We look forward to the spring election for two reasons: 1) presidential elections always have a higher turnout, and 2) students will be more aware of the new voting system by then.

What went well?

The planning and programming of the system went very well. We were able to produce a fully functional, secure, and stable system in a month with one programmer working 20 hours per week. This included the database schema, procedural design, coding, and testing. As a testament to the very careful and thorough design process, there were actually no bugs or security holes uncovered during public beta testing. There was one usability issue suggested—the ability to confirm who a voter voted for before doing a final submit—but there were no actual deficiencies or incorrect operations discovered. The error messages voters receive upon a failed login were improved, however, to make them more informative.

What could have gone, or been done, better?

As mentioned earlier, we did have some problems with the data provided by PolarExpress. In addition to missing or incorrect information, some records did not list a library card number, and there was one record that was missing a birth date.

While we were hoping for higher turnout, we were pleased with the increase.

Another thing I would have liked to be different was the amount of time allocated to the project. We had one day over a month from the time we were given the official go-ahead until the polls opened. This made for a very tight schedule, as well as a bit of uncertainty, every now and then, as to whether or not we could make the deadline. One evidence of this lack of time is the current lack of an administrative interface: ballot and survey questions and answers currently must be added to the database by hand. For this election, we actually linked to the database via ODBC drivers for MySQL and made forms in Microsoft Access. This greatly simplified the entry of data, but an administrative interface is needed, and will be worked on during the next semester. There is, however, an election reporting module.

The Future

The future of online elections, both for UAF and the University of Alaska statewide is an exciting prospect. University of Alaska Information Technology Services (ITS) has expressed interest in integrating online voting capabilities into the currently-in-planning "My UA" system, a one stop web portal for students, staff, and faculty that will allow them to access all the information and functions pertinent to their careers at UAF. We (ASUAF) were told that they (ITS) were very interested, but needed a proof-of-concept, which we have now provided.

Another possibility, at least for the short term, is the offer that has been made by the Registrar's Office here at UAF. Now that we have a successful proof-of-concept, they have mentioned the possibility of having every student that registers for classes also register for an ASUAF account. This would ensure that every student has a username and password on our system, and would obviate the need for PolarExpress data extracts. However, this would introduce the complexity of checking all accounts every year for validity. In this case, my preference is to fold online voting into My UA.

Updates – Version 1.2

As does any project, this one went through updates and improvements. Not so much in the core software—there were no new functional requirements, although a few changes were found to be needed—but more in the supporting software. In the next round of elections, a few changes were implemented.

Linux Distribution and Packages

When we upgraded our server to Mandrake 9.1 in the summer of 2003, the decision was made to go with the Mandrake supplied packages instead of compiling from source. This ending up presenting no problems, and reduced administrative load significantly.

MySQL

We are now using MySQL 4.0.15. It was a drop-in replacement for 3.23.x, requiring no changes to the code. It also got rid of the "slow queries" we had never managed (or bothered) to find. Also, we use a version of MySQL compiled from source due to the need for a small source-code change to work around a bug in Microsoft Access.

Apache

Our web server was upgraded to Apache 2.0.47. After the requisite configuration for the SSL certificates, everything worked as before. The Mandrake source package for Apache was needed as we had to recompile suEXEC to change the directory in which it allowed execution (this is hard coded at compile time).

Session ID

With our limited time line during initial roll out, we did not have time to do extensive testing, even though we did test all the functionality of the system. Load testing was one thing that suffered, to the point of not being done. When the system was used under moderate load (in subsequent elections), it was discovered that OpenSSL/mod_ssl would sometimes generate the same SSL_SESSION_ID for two different requests. This was probably due to the request hitting the web server in the exact same second, thus the seed being the same (we never did investigate). This happened enough that another solution was needed. In one of the custom utility libraries already written, there was a random string generator. It looked like this:

```
sub gen_rand_string
        {
        ########################################
        # Sub Name: gen_rand_string
        # Purpose: generate a random string
        # Parameters Passed In: int($length)
        # Values Returned: string($rs)
        ########################################
        my($length) = @_;
        my $rs = ""; #password string
        my $x = 0; #counter
        my $num; #random number
        my @chars = qw"A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z a b c d e d f g h i j k l m n o p q r s t u v w x y z
1 2 3 4 5 6 7 8 9 0"; #All one line

        #Run srand() unless it's been done already
        unless($Common::RAND_OK)
                {
                srand(time() ^ $$ ^ unpack("%16L*",
`/home/asuaf/util/ps axww | /bin/gzip -fc`)); #All one line
                $Common::RAND_OK = 1;
                }

        for($x = 0; $x < $length; $x++)
                {
                $num = int(rand(63));
                $rs = $rs . $chars[$num];
                }

        return($rs);
        }
```

This generated random strings of sufficient quality that we never ran into a duplicate session ID again. Since we are now using upper and lower case letters, as well as number, we now have 62^{64}, or $1.4*10^{115}$ possibilities.

Question Ballots

Due to the glitches in the data we were given, as well as some students simply preferring not to use online ballots, question ballots were brought back. The student may fill out a paper ballot, as before, placing it in an envelope bearing their name and student ID. Then, once the election is

done, we run a report of all those who voted, and make sure they did not also try to submit a paper ballot.

End Notes

[1] Perl is interpreted in the sense that the source code is stored in a plain text file, and the plain text file is executed as a script. It is compiled in the sense that upon execution, the script is compiled in memory, and it is the compiled version that is run from then on: there is no further runtime interpretation of source code.

[2] Lightweight Directory Access Protocol

[3] YP (Yellow Pages) is what Network Information Services (NIS) used to be called until it was discovered there was a trademark on the term Yellow Pages®.

[4] Technically, `ev_ballot_items.election_id` could be linked to `ev_vote_config.current_election_id` and/or `ev_election.id`, however, this relationship is not drawn on the diagram (below) to emphasize the two categories into which the tables fall.

[5] "Already Voted" is defined as voting on the official ballot, as well as filling out the survey (if there is one).

[6.] The `$PATH_INFO` variable provides a mechanism whereby a script can actually masquerade as a directory. For example, given a request of `http://www.example.com/main/support/widgets`, "main" could actually be a script which, when it receives a request with `$PATH_INFO` equal to "`/support/widgets`," will retrieve the files (or database records, etc) which define the layout of the support area, and the content of the widget support area.

Bibliography

Dyck, Timothy. (2002). Server Databases Clash. *eWeek* 2002-02-25, retrieved 2002-10-20 from http://www.eweek.com/article2/0,3959,293,00.asp

OpenSSL Team. (2002). OpenSSL: The Open Source toolkit for SSL/TLS. OpenSSL web site, 2002-10-20, retrieved 2002-10-20 from http://www.openssl.org

mod_ssl Team. (2002). mod_ssl: The Apache Interface to OpenSSL, mod_ssl web site, 2002-10-20, retrieved 2002-10-20 from http://www.mod_ssl.org

More Information on Resources and Software Mentioned

Linux Operating System

http://www.linux-mandrake.com/en/
http://linux.com/
http://www.linux.org/

Apache Web Server

http://httpd.apache.org

MySQL Database Server

http://www.mysql.com

The Perl Programming Language

http://www.perl.com
http://www.perl.org

OpenSSL

http://www.openssl.org/

mod_ssl

http://www.modssl.org

Change Log

Version 1 (2002-12-05)

Initial Release

Version 1.1 (2002-12-08)

Added *Change Log* and *To Do* sections
Minor grammatical changes
Fixed a few typos
A few explanatory phrases added here and there
A few phrases restructured for clarity

Version 1.2 (2004-07-07)

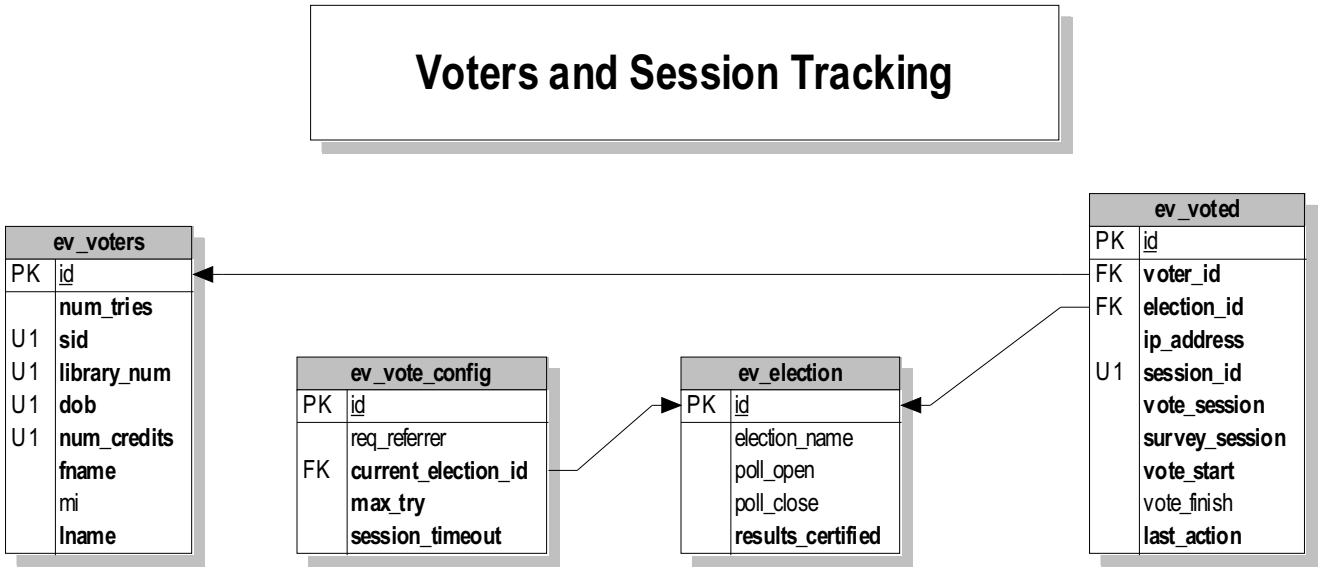
Minor formatting changes
Additional explanations
Updates and clarifications
Minor grammatical changes
Typo fixes
Formatting changes
Added Updates section

To Do:

Add a code commentary of all the code used in the voting system

Database Commentary

The tables in the voting database are divided up into two sections: The group of tables for Voter and Session Tracking, and the group for Candidates, Questions, Votes and Answers. The structure of the database is tied together via a relational model. All primary keys are auto-increment, unsigned, four byte integers, except for `ev_vote_config.id` since that table will only have one record. All tables also have an automatic MySQL `TIMESTAMP` type field to facilitate attaching the tables to a front end such as Microsoft Access.



For system configuration and information, the system relies on `ev_vote_config` and `ev_election`.

The `ev_vote_config` table has a single record which holds the system configuration information. The `req_referrer` field is not used, but is in place should the check for referrer be implemented in the future. The `max_try` field dictates the number of login tries a voter may make before they are locked out, and `session_timeout` is the maximum time a session may go without activity before it times out. `Current_election_id` points to the election that is currently being run. As the system is designed, only one election can be active at once. There can be multiple elections configured and ready to go, but only one may be running at a time.

The `ev_election` table holds information about elections. `Election_name` gives a display name to the election, such as "ASUAF 2002 Fall General Election." `Poll_open` and `poll_close` determine, as is logical, when the polls open and close. The `results_certified` field is not currently used, but is there for future use when we may want to have an auxiliary script that will display the results online after they have been certified.

`Ev_voters` and `ev_voted` keep track of users and their sessions.

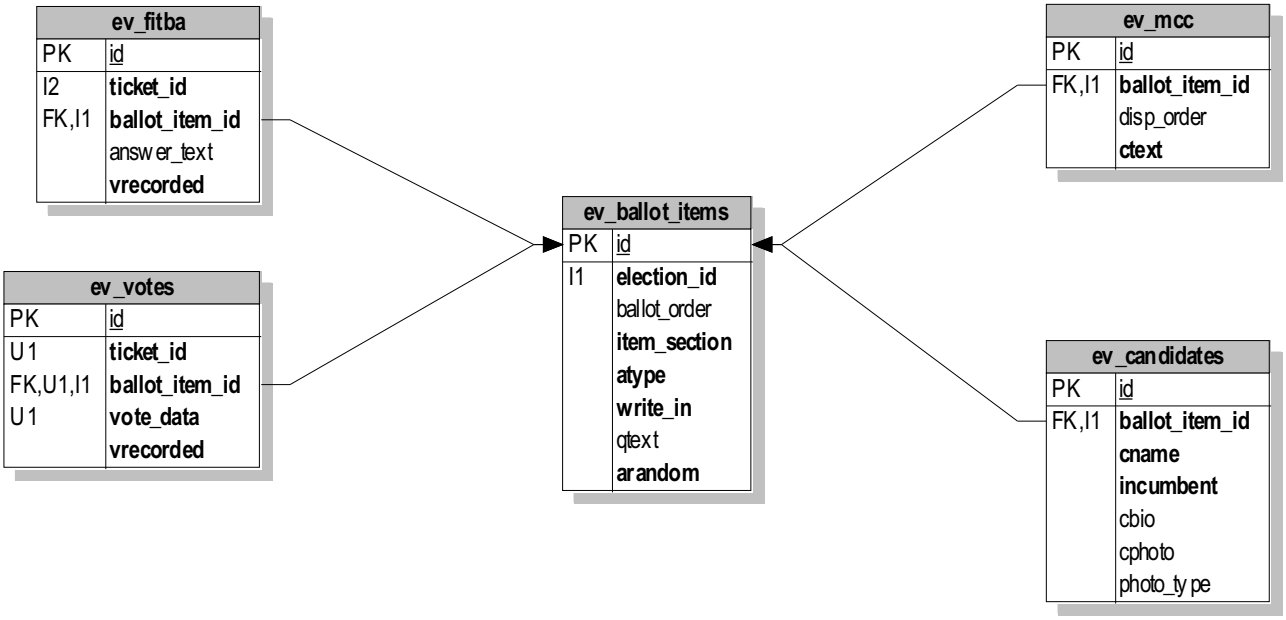
`Ev_voters` contains the user information obtained from PolarExpress. The `num_tries` field records the number of times a user tries to login with incorrect information. The personal information we check is contained in `sid` (student ID number), `library_num` (also known as their PolarExpress number), `dob` (date of birth) and `num_credits` (number of credits in which the student is currently enrolled). The fields for first name, middle initial, and last name (`fname`, `mi`, `lname`, respectively) are not currently used, but the data from PolarExpress is placed there since it is given to us. In the future we may have a personalized thank you message printed after the ballot or survey has been filled out.

The `ev_voted` table keeps track of who has voted, and the time of their last action. `Election_id` keeps track of the election in which the user has voted. While the likelihood of it being implemented is low due to the fact that `ev_voters` must be cleared and re-imported before each election, it allows records to be kept of who has voted in past elections. `Ip_address` is a four byte, unsigned integer that stores the IP of the voter. The dot quad notation is converted to a four byte integer by the MySQL function `INET_ATON()`. `Session_id` is the field in which we store the `$SSL_SESSION_ID`. `Vote_session` and `survey_session` are enumerated fields with possible values of Open, Closed and Aborted. The sessions are open before the voter votes in those categories, closed after he votes, and aborted if they were open a half hour after the polls closed. `Vote_start`, and `vote_finish` record the times the session was opened and the time the voter submitted the ballot, respectively.

`Last_action` records the time of the voter's last activity. If he attempts an action, and this value is more than `ev_vote_config.session_timeout` minutes in the past, the session has expired.

The ballot and survey questions and answers are kept in `ev_ballot_items`, `ev_mcc`, `ev_candidates`, `ev_fitba`, and `ev_votes`.

Candidates, Questions, Votes and Answers



`Ev_ballot_items` stores all the questions for each election, keyed on `election_id`. The `item_section` field is an enumerated field which has the possible values of Senator, President, Regent, SCPSE (Student Commissioner for Post Secondary Education), Referendum, and Survey [Another category was recently added for Concert Board Nominee]. `Ballot_order` determines the order in which questions will be displayed. Across sections, this will have the side effect (if the questions are numbered uniquely) of ordering the sections as well. It is usually used, however, to determine the order of questions within a section that has more than one question, e.g. the survey or referendum sections. `Write_in` is an integer field which determines if a field has a write-in box in addition to the other choices. `Atype` is an integer field which determines the type of answer to be given to the question. If atype is 0, and `write_in` is 0, then the input type is a text box; if `write_in` is 1, then the input type is a text field. If `atype` is 1, then the choices will be printed with radio buttons (only one choice available). If `atype` is more than one, then the choices will be check boxes, with `atype` dictating the maximum number of choices. For radio buttons and check boxes, `write_in` (1 or more) determines the number of text fields that are printed along with the choices already in the database. The field `qtext` contains the text of the question, and `arandom` determines whether

the answers should be displayed in random order (`arandom` ==1) or in the order they are dictated (by `ev_mcc.disp_order`) in the database (`arandom` ==0).

The `ev_mcc` table stores the choices for multiple choice questions. `Ballot_item_id` determines to which question the choice applies, `disp_order` determines the order in which the answers will display if `arandom`==0, and `ctext` is the text of the choice.

`Ev_candidates` stores information about candidates. `Ballot_item_id`, again, points to the item to which the entry refers. `Cname` contains the name of the candidate(s); `incumbent` is a true/false field with their incumbent status; `cbio` contains their biography. `Cphoto` and `photo_type` are for the future use of storing pictures of the candidates in the database.

`Ev_votes` contains the selected answers to the multiple choice questions, such as candidates, referendums, and surveys. The `vote_data` field contains the `ev_candidate.id` or `ev_mcc.id` of the answer selected. `Ev_fitba` contains write-in answers for elections and surveys. `Answer_text` contains the name or text that was filled in for the answer. Both tables have `ticket_id`, `ballot_item_id`, and `vrecorded` (time the vote was recorded).