i.d. miller
6114 La Salle Ave Box 543
Oakland, CA 94611
http://idmiller.com

Implementing a Psychological Survey using Open Source Software.[1]

http://iandennismiller.com/media/pdf/oss_survey(2008).pdf

Abstract

An online social psychological survey was implemented using a variety of Open Source Software products. The specific technologies, the way in which they were combined, and the challenges are discussed, as well as the requirements that were fulfilled. The scientific results of this work, as well as other methodological considerations, will be published with the following approximate citation:

Page-Gould, Mendoza-Denton, and Tropp. “With a little help from my cross-group friend.” Journal of Personality and Social Psychology. November, 2008.

1 The 2006 version of this paper is now deprecated, but it is available from the following address - http://iandennismiller.com/media/pdf/oss_survey_deprecated(2006).pdf

Table of Contents

Introduction

I was hired in July of 2003 to program an online survey for a longitudinal social psychological experiment being conducted at the University of California at Berkeley by Dr. Elizabeth Page-Gould, Professor Rodolfo Mendoza-Denton, and Professor Linda R. Tropp. The timeline for the project was fairly compressed and inflexible, and the available resources were scarce. Taken together, these interesting constraints influenced the entire direction of the work towards Open Source Software because:

- Open Source Software (OSS) is free on its surface; OSS is a tradeoff between budgeting for labor and budgeting for commercial software
- commercial software is more difficult to modify internally (it is "closed source")
- commercial software is generally geared towards projects with high-performance requirements, which was overkill in this case, to the extent that the timeline might actually suffer for it
- I was personally familiar with the OSS offerings because my undergraduate education favored OSS as a learning tool, because of the visibility of the source code

This paper is organized into two sections: the features required by the project, and the technologies used to implement those features. The technologies have since improved, but the features that are fundamental to any survey are orthogonal to the technologies that might implement them. In short, pencil and paper (old technology) will still suffice, and can even benefit from the discussion of survey features, but the technology currently available gently suggests that a better approach exists.

Technologies

For the purposes of this discussion, technologies are both hardware and software. From a planning perspective, a unit of technology is a “black box” whose function is understood without necessarily understanding how it functions. The automobile is a prototypical black box technology: I expect the car to stop when I hit the brake, but I don't worry about how the anti-lock braking system detects and reacts to road conditions.

Planning the software architecture of an on-line survey follows the pattern of many web applications, which is frequently called Model-View-Controller (MVC). The MVC pattern, applied to web applications, generally calls for the following technologies: on the client computer is a web browser, and on the server computer is a web server, a database server, and the application that controls it all.

The term LAPP Platform (or LAMP) is commonly used in the Open Source web application literature due to its relation to the MVC pattern, where LAPP stands for Linux, Apache, PostgreSQL, and Perl. The LAPP platform of software technologies, which will be described in detail, provide a web server, database server, and application programming language, all of which are open source. All of these technologies are built on that most-general technology: the Intel x86 processor and the “generic PC.”

In discussing the technologies involved in this on-line survey project, I will work from lowest level to highest, providing a peek at the insides of these “black box” technologies as they related to the on-line survey project.

Hardware

I used two physical computers in this project: a development machine, which I administered, and a production server, which was controlled by the lab that performed the research. Hardware is still an issue for any lab or project, and it is possible to create higher demands than your hardware can sustain. Unfortunately, it is difficult to provide any rule-of-thumb in estimating hardware requirements because it is depressingly easy to code inefficient web applications.

Consider the following requirement: up to 8 users will simultaneously access a database, each submitting approximately one page of data per minute, which amounts to 8 pages per minute. This means that, on average, the survey must be able to sustain one page every 7.5 seconds. For perspective, Amazon manages tens of thousands of pages every 7.5 seconds, and this 8-person survey only has to handle one page in that period of time.

In terms of hardware, of course, Amazon has thousands of computers, and you might only have a single, generic PC from 1997. I'll go out on a limb and conservatively estimate that the 1997 PC probably won't suffice for our 8-user requirement, as meager a requirement as it might be. Also, keep in mind that decade-old hardware suffers from other aging problems, too.

In practice, my hardware consisted of two machines for implementing this survey. Here are the exact specifications:

i.d. miller “development” machine:

- Operating System: GNU/Linux
- Processor: Intel Xeon 1.8GHz P4 (2 processors)
- RAM: 1GB (512MB, 2 DIMMS)
- Hard Drive: 36GB SCSI 160

UC Berkeley “production” machine:

- Operating System: Windows XP SP2
- Processor: Intel P4 2.53GHz (1 processor)
- RAM: 512MB (1 DIMM)
- Hard Drive: 60GB ATA 100

The hardware was more than adequate for the needs of the survey. However, such a setup had interesting consequences, not the least of which was the difference in operating system, which I will discuss in greater detail.

Assuming the hardware is utilized well by the software, I will venture some rule-of-thumb estimates (anecdotally taken from personal experience) for the 8-user survey:

- newer is better: Intel P2 400MHz is scraping by, Intel P3 800MHz is comfortable, Intel P4 2.5GHz is quite comfortable.
- no matter the age of the computer, maximize RAM: 128MB is scraping by, 256MB will be adequate, and 512MB will be comfortable.

GNU/Linux

By now, the Linux kernel may need no introduction, as it appears in the media with increasing frequency. Linux, or more specifically, GNU/Linux, is the combination of the Linux kernel with a wide range of tools that approximate a Unix environment. The kernel, as well as the GNU tools that make it usable, are all Open Source products. The nature of the Linux kernel is to serve as an interface to a computer's hardware, and in this case, the less exotic the hardware, the better.

Hardware compatibility issues did pop up occasionally on the development server, although this did not affect the project timeline. To illustrate, the Linux kernel (in 2003) did not support certain error codes that were generated by the processors in the development machine. Because the hard drive was physically installed in such a way that it restricted airflow to the processors, the processor temperature exceeded its threshold, and generated a “Machine Check Exception” that the Linux kernel was unable to handle, which resulted in a very nasty, sudden machine shutdown. The solution to this problem was twofold: moving the hard drive to restore airflow, and waiting for the gradual update of the Linux Kernel, which can now respond to temperature events by lowering the processor speed.

The common criticism of GNU/Linux comes from the perspective of the desktop user experience. This is largely irrelevant in the case of online web surveys. In fact, considerable benefits come from not running a graphical desktop at all, including better security, more predictable operation, and better utilization of memory. The desktop argument aside, GNU/Linux supports a suite of common server tools for making web applications: Apache, Perl, and Postgres, all of which are discussed in detail.

GNU/Linux is an excellent operating system, but as the anecdote warns, unexpected interactions and fringe operating conditions can result in serious troubleshooting. In my experience, this is easily avoided by using mainstream, modern hardware. GNU/Linux performs well as a dedicated server operating system, and is a good platform for building web applications.

When people talk about Linux, they often discuss the wide variety of Linux “distributions”, which are also referred to as flavors. To briefly comment on Linux distributions, some are tailored to new users and others to experts, so it is worth it to research before making a decision. During this project, I used Gentoo Linux, which is only appropriate for advanced users, but I currently use Ubuntu Linux, which is much more accessible. I would recommend Ubuntu to OS X users (the underlying operating system is quite similar), and I would even venture that the switch for Windows XP users can be managed.

Back in the real world, labs can't often afford a technician who knows Linux, and may not be lucky enough to have a volunteer with functional Linux administration skills, either. Although practically all computers can run Linux, a measurable expense manifests through the support of Linux. If “computer support” isn't literally in the budget, it implicitly appears under other headings as “wasted time.”

Windows XP

Back in the real world, labs can't often afford a technician who knows Windows XP, and may not be lucky enough to have a volunteer with functional Windows XP administration skills, either. Although practically all computers can run Windows XP, a measurable expense manifests through the support of Windows XP. If “computer support” isn't literally in the budget, it implicitly appears under other headings as “wasted time.”

In contrasting these two statements about “wasted time,” the difference is clear: computers are difficult to use, and that value can be measured in terms of support time. I assert that support time is partially a function of troubleshooting knowledge and training. Windows XP (often called XP for short) is a monster to learn, but it so happens that a lot of people in the academic world are reasonably well-trained at XP. As a desktop operating system, it is easier to use than most alternatives. Therefore, Windows XP is probably the most common operating system used in academic labs, and that is the kind of computer you can expect to find lying around.

In the case of the on-line survey project, one requirement was that the survey be hosted on a machine running XP. This requirement is not too difficult to derive: for the purposes of data security, the machine must be kept physically secure, and the lab already had a locked room to store it in. Furthermore, without budgeting for a dedicated machine, an existing machine would have to be reused, and where else would that machine be but in the lab? Unfortunately, from the Unix perspective, XP is not well suited to be a server operating system.

Cygwin

My preference is to develop for GNU/Linux, but the project required Windows XP. I chose to bridge this gap using Cygwin, which is a Windows XP tool that provides a similar experience to Linux. Cygwin also serves as a gateway into a world of pre-compiled Linux applications that run within the Cygwin environment. To relate this back to the LAPP platform, it is possible to swap out Linux for XP, because Apache, Postgres, and Perl are all quite functional under XP with Cygwin.

This level of functionality is provided for by competing Open Source projects, too, such as Indigo Perl or XAMP. In the end, there is only one requirement: that the computer runs Apache, Postgres, and Perl. The Operating System is, in a sense, secondary. However, committing to XP and Cygwin has its consequences, both positive and negative.

On the negative, this means you must monitor yet another mailing list to watch for Cygwin security updates, and you must act on them. On the positive side, though, Cygwin really behaves like Linux in many ways, so remote system administration becomes possible (via OpenSSH). Over the course of the survey's lifespan, the need to remotely administer the web server will inevitably arise, and fixing it faster is always better. It is a different process to remotely administer Indigo Perl and XAMP, and I personally feel less comfortable with it.

A major difference between XP and Linux that Cygwin underscores is the process for automatically launching an application at bootup. Also, Cygwin appears less like Linux the deeper you dig, so depending on the requirements of your web application, you may find major architectural differences that are unexpectedly painful to code around.

For most on-line surveys, this won't be a problem, but consider the following: as part of a data encryption scheme, I proposed using a certain algorithm that depended upon a specific library of code containing exotic math functions. While I could easily install these math functions on Linux, I ran into deep architectural difficulty with Cygwin and XP. In short, a different encryption approach was required because Cygwin wasn't similar enough.

Apache 1.3 web server

The Apache web server (often called just Apache) is a special Open Source project. Beginning as just a web server, through the explosion of the Internet, the project has blossomed into the Apache Foundation, which is an incubator for dozens of other projects now, including significant technologies like the Apache Tomcat server. The Apache Foundation is very alive as a community, and the Apache web server appears to have a solid future ahead of it.

In building a web application, the web server manages the connection between a client (who is generally using a web browser) and the web application (which, in this case, is a survey program). Apache performs admirably in its web server capacity by providing accelerated interfaces for many programming languages, including PHP and Perl. In communicating with clients (including potentially malicious clients), Apache has historically been quite secure, with infrequent but timely security updates.

As with any web server, security will be a major issue. Principally, a web server is software running on your machine that allows a remote user to control it. This introduces all sorts of complications, and it becomes extremely important to limit the actions a client can perform. Apart from your Hollywood-style “hacker” scenario, in academic research, a potentially serious problem is exposing sensitive data. Specifically, the layers of interaction in any moderately complex web application potentially introduce new avenues for an information leak, which means, for example, that a completely secure web application can be undermined by an insecure web server configuration.

The web server is among the most visible security components in an on-line survey, but it is by no means the end of the story on security. In the same way that a secure application is undermined by an insecure web browser, a secure web browser is undermined by an insecure operating system. As far as the OS is concerned, Apache installs on all flavors of Linux (or comes pre-installed). In the case of XP, it is a mixed bag, but as was discussed before, Cygwin, XAMP, and Indigo Perl all present options.

PostgreSQL

PostgreSQL is an opensource database server. While it faces fierce competition from MySQL, a peer in the Open Source database arena, I have backed PostgreSQL for several years due to a single advantage it alone possesses: ACID compliance (which stands for Atomicity, Consistency, Isolation, and Durability). Whereas MySQL can fall into a corrupted state, to be recovered later by their recovery software, PostgreSQL will never enter such a state in the first place. To accomplish this, PostgreSQL has historically showed slower performance than MySQL, which is an important factor in the commercial sphere.

However, because the performance requirements were fairly modest, and the cost of losing any data was immense, the decision to pick PostgreSQL over MySQL seemed obvious. PostgreSQL installs easily enough using Cygwin, and is generally simple to install under Linux, as well.

A number of unique factors affect the database aspect of an on-line survey. A database server supports a finite number of simultaneous connections, but the behavior of Apache is to hang on to connections longer than is necessary, which can unnecessarily impact performance. It is also a non-trivial task to design a functional database table for the needs of a given on-line survey.

Perl 5.8

Perl is a programming language. Syntactically, it is pretty similar to C++. In execution, Perl is an interpreted language (as opposed to compiled). It is modern, in the sense that it supports object orientation (ugly, but true) and it provides memory management and garbage collection. Finally, Perl is very social, perhaps unlike anything before it. Through CPAN (the Comprehensive Perl Archive Network), a massive library of code is available for reuse. Although it is at first difficult to reuse other people's code, the payoff is massive.

Returning to the LAPP platform, the final P stands for Perl. The acronym is clever, but in practice, Perl can be swapped out for PHP, Python, Ruby, and any number of other Open Source programming languages. Each language has its fans and detractors, although in the web sphere, there is particularly high use of PHP. Personally, I prefer to think of an MVC application in Perl terms, although there have been projects to adapt MVC to PHP, for example.

As languages go, Perl has a notorious reputation for being ugly and illegible. This is easily true, and just as easily false. From experience, I know that it is possible to create elaborate byzantine structures and labyrinthine tangles of code, complex enough that they work but you have no idea how. Also, from experience, I know that when your code is that complex, you ultimately have a poorly organized understanding of the underlying concepts. I will confess that the on-line web survey suffered from a touch of the labyrinth and deviated rather far from the MVC pattern. These often go hand-in-hand.

Perl interacts very well with Apache and with PostgreSQL. In the case of Apache, the mod_perl project provides the ability to directly integrate Perl code into Apache, resulting in a massive performance payoff. PostgreSQL is accessed from Perl through the DBI module, where Perl DBI can improve performance again through mod_perl and Apache. In short, Perl is an excellent language for coordinating web server and database server activities because it integrates deeply with both.

Web Browsers

The web browser is generally the target of a web survey, or rather, it's the person who is using the web browser. Part of the challenge is in ensuring that nothing interferes with this process, so that everyone sees the same fundamental thing. Of course, this is notoriously difficult.

Much of the difficulty started with the so-called “browser wars” of Netscape Navigator versus Microsoft Internet Explorer. An international standards body, the W3C (World Wide Web Consortium), established a series of specifications for HTML (Hyper-Text Markup Language), upon which the so-called World Wide Web has been built. Microsoft and Netscape both violated the standards, creating a rift in the HTML language, and an endless nightmare of compatibility problems for web designers as a whole, but for on-line survey engineers specifically.

Adding to this mess is confusion about what, exactly, a web browser is capable of. To answer the first question that comes up, millisecond-scale reaction times are not possible to record using a stock browser. Experiments that purport to record millisecond-scale reaction times must rely on Java applets, which can be embedded in the browser, but which still cannot reliably guarantee such accuracy.

Although it is difficult to consistently display web pages properly, and reaction times cannot be recorded for an on-line survey, it is still possible to use the web browser to collect certain, meaningful kinds of data.

Web browsers also play a key role in encrypting communication. By serving as one endpoint in an SSL-encrypted channel (with the Apache server as the other endpoint), the web browser helps to minimize the chances of third-party eavesdropping.

Features

The vocabulary varies on this topic, ranging from "requirements" to "features" to useful frameworks such as "Use Cases", but for the purposes of this paper, "Features" are the things that the software accomplishes. The primary feature of a minimally functional survey is that of data collection; participants enter data, and the survey records it. For consistency in vocabulary with other work, "participants" are the same as the deprecated term, "subjects."

Data Tables and Recording Data

To accomplish the survey's primary objective, participants responses were recorded in a database. There are two primary paradigms that will accomplish this: the "spreadsheet", and the "journal". A spreadsheet contains a column for each variable to be recorded and a row for each participant, and each data point that is recorded is entered at the appropriate row/column coordinate in this matrix. A journal, on the other hand, is a running list of each data point entered, and it contains only four columns: the user identifier, a timestamp, the variable name, and the recorded value of that variable.

Databases consist of a collection of tables, where each table consists of rows and columns. Although this is similar to a spreadsheet on its surface, a database requires a schema that will rigorously define the data that will be recorded in each column. Whereas Microsoft Excel allows columns to be cut-and-pasted, a database can only be altered by updating the table schema.

In the scientific survey world, a rigorous survey has a corresponding “data dictionary” which will clearly identify the variables being recorded, the names of those variables, and the type of data that is captured by each variable. This corresponds directly with the database table schema, in the case that a “spreadsheet” paradigm is used to organize the data.

The danger inherent in the spreadsheet paradigm is that changing survey requirements will impact the schema. In this particular instance, the survey was being altered at the same time that it was being implemented as a web application, and each change would require the schema to be altered. Because there were more than 1,000 variables, this task alone could be prohibitively time-consuming.

The journal paradigm is one way to address the complication introduced by rapidly-changing requirements. In this case, the table schema is trivially composed of four columns, which again are the user identifier, a timestamp, the variable name, and the recorded value of that variable. The variable name is not part of the table schema, but is instead a data point contained within the table.

Because it is much easier to alter data than it is to alter the schema, the journal can rapidly incorporate changes to the data dictionary without modifying the schema. Also, because data points are individually recorded and timestamped, this time information is available for analysis, and can be helpful for data cleaning.

The primary danger to the journal paradigm is that there is no statistical software package that can natively process such data. Therefore, the journal must be transformed into its corresponding spreadsheet form, so that it can be imported by the relevant software. This extra step is performed after all data has been collected, and can be referred to as “exporting” the data by reconstructing the column headings and creating a comma-delimited or tab-delimited representation.

The end result is that both paradigms will capture the same data in a manner that is appropriate for statistical analysis. After using both methods on a variety of other survey projects, I recommend the use of both, simultaneously, and if only one is to be used, I recommend the spreadsheet paradigm. Ultimately, this recommendation is based on the importance of keeping the database in sync with the data dictionary, and for the purposes of rapidly verifying the the database is actually recording the required variables. This requires the survey to be completely designed before implementation begins.

User Identification

Each data point that is recorded must be associated with a unique user identifier for the purpose of analysis. Each statistical hypothesis consists of dependent and independent variables, where the user identifier is generally an independent variable, and is therefore crucial to any statistical test.

For this survey, participants were assigned a unique numerical ID number that would be theirs for the duration. This ID number is intended to separate the identity of the participant from their associated data through the use of a one-way table that maps IDs to actual identities. Theoretically, it should be impossible to determine the identity of an individual's data without access to the one-way table.

There are a number of guidelines for creating these identifiers:

- IDs must be unique. If it is ever the case that two individuals are given the same identifier, their data will be confused and potentially ruined.[2]
- IDs should be difficult to predict. Consider the case that the first user is assigned ID #1, and the second user is assigned ID #2. A number of malicious actions are now possible, ranging from using another user's ID to reconstructing personal identifiers based on the sequence in which those individuals participated in the survey.
- IDs should be sparse within the ID space. If you will have 100 users, and if you allocate the numbers 0-99 for identifiers, then even if these numbers are randomly assigned, it will be the case that a malicious user could easily guess another ID since 100% of the ID space is filled. Better than IDs 0-99 would be 0-999, where only 100 numbers would be used out of the available 1,000. Here, only 10% of the ID space is used, and it would take 10x longer to randomly guess a valid ID.

The topic of user identification will be raised later in this paper, because certain practical constraints further influenced the theoretical concerns listed above.

2 It is possible to recover from this situation if a journal paradigm is used to record the data, because timestamp data can be used to disambiguate between the individuals.

Security and Privacy

A variety of sensitive and personally identifying data points were recorded in the database over the course of this survey. One of the data points was a contact email address, which would be used to follow up with participants if they requested to receive a copy of the finished publication. Clearly, the existence of this data point would allow anyone to immediately associate data with an identity, due to the identifying nature of email.

The solution I employed was to use strong encryption, such that certain data points would be impossible to read without access to the encryption key. In retrospect, I do not recommend this because it added complexity to the project, and I would instead suggest using a separate table whose only purpose was to record email addresses. This table must not contain the user's ID or any other item (e.g. timestamp) that could be used to associate data with an identity.

In some cases, strong encryption might be justified, but there are inherent risks. The server must contain one copy of the encryption key, in order to perform encryption on the data before archiving it. The principal investigator also has access to this key, and can therefore decrypt the data. However, any unauthorized access to the server can potentially compromise the key, so the use of third-party encryption hardware is the best practice.

Dynamic Elements

Dynamic Elements are elements appearing in the survey that are determined at run-time, or "on the fly." This may be driven by the condition the participant is assigned to (randomly or otherwise), or by answers the participant previously provided. In the case of this survey, both were the case: survey condition was determined by the background of the participant, and the names of participants' friends were reused a number of times throughout.

An example of a dynamic element is to create a survey item that contains a "placeholder," which is blank until actually viewed by the participant. Upon viewing, the placeholder is replaced with, for example, a word representing their ethnicity, as in:

"Imagine you are out tea with a ______________ friend"

which becomes, depending on the condition:

"Imagine you are out to tea with a caucasian friend" (though this can just as easily be "latino/latina" or any other scientifically indicated phrase).

Although it might appear trivial to accomplish this, consider the following example, where a participant has previously indicated that their friend is named Alfred:

"How close do you feel to ___________?"

which becomes:

"How close do you feel to Alfred?"

In this case, the friend's name has been recorded in the database, so it must be looked up in the database in order to be displayed. Depending on the volume of concurrent users that the survey is actively communicating with at that moment, the total number of entries in the database, and the design of the database, this lookup to retrieve the name can take a non-negligible amount of time.

It is a matter of user experience that this retrieval time be minimized, because in the extreme, this could potentially alter the experience of the participant in an experimentally noisy manner. As the number of users increases, the delay might increase, causing participants who take the survey later to experience a longer delay than those who took it earlier.

The solution is to properly design the database with this in mind. Because this was an issue of this survey's table design, a feature of Postgres called an "index" was used, which has the effect of pre-sorting the data tables based on a particular field. Because users were uniquely identified by an integer, the table was kept in a pre-sorted state on the basis of this identifying integer.

To take advantage of this, database entries were first filtered by user identifier, which had the consequence of capping the total number of fields to be scanned for the friend's name. This was verified by including the "render time" of a page in special administrative debugging output, where render time is the amount of time it takes the page to be displayed.

Loops and Forks/Skip Lists

Survey designers are accustomed to allowing for multiple paths through the survey, based on participant responses. A common example comes from television market research, whereby a phone survey ends immediately if the individual does not own a television.

One requirement of this survey was to first prompt the participant for a list of their friends, and then for each friend to ask certain questions. The challenge, then, is to use the database for read/write access, to enable the survey to “look back” at previous answers to figure out what to present to the user now.

A unique challenge faced by web-based surveys is the “statelessness” of the HTTP protocol, which is the fundamental language of the world wide web[3]. This stems from the fact that web browsers and servers communicate through bursts of information. This can be contrasted with a phone call, where the line is kept open for the duration of the conversation, and it can be assumed that as long as the line is open, the same two people are talking. HTTP does not “remember” anything about previous conversations, so it is unable to tell the web server that a given communication logically follows a previous communication.

The implication for loops and skip lists is that the web application must include additional logic to remember where the user is, within the survey. This was accomplished indirectly, by continually visiting the database to read the list of friends, and to determine how many friends have already been recorded. On this

3 Consider a typical web address such as http://www.google.com – here, http is explicitly indicated as the protocol to use for communication.

basis, it was possible to conclude which name should be presented to the participant. As was mentioned in the previous section about dynamic elements, this technique creates significantly more work for the web server, because it must repeatedly execute a query that may be fairly complicated.

Rather than using the data table to indirectly remember the user's location within the survey, it is also possible to create an explicit “state table” that temporarily stores the user's progress through the survey. This table can be crafted to specifically hold “state information” such as the current page in the survey or the list of the participant's friends. However, the risk in this approach is that the state table may become out of sync with the data table, which could easily happen if the participant uses the web browser's “back button” to return to a previous page.

Regardless of the technique used, it is essential to maintain such state information in order to implement loops and skip lists within the survey.

Multiple Sessions and Obtaining Access

This longitudinal survey consisted of 15 sessions for each individual, which were spread out across one month. This time-scale created a unique problem for the user identifiers, because they would need to be used by multiple parties over a long period of time. Participants would need to access this survey from the lab, but also from their own homes, so a mechanism had to be devised for them to obtain access to the survey.

To support these requirements, a login page protected access to the survey until a valid ID was entered. Because participant apathy is a serious problem at UC Berkeley, this system needed to be trivially simple, since it could not be assumed that participants would make even a second attempt to log in. Ultimately, this grotesque laziness would result in data loss, at great expense to the research. An additional requirement was that participant need not know their unique ID – this information would be managed by research assistants.

The survey was available from a URL that looked like http://some.berkeley.server, and this URL would be emailed to participants over the course of the month for their continued participation. To facilitate logging in without ever knowing the ID, a special suffix was appended to the URL that would bypass the login page by automatically entering the ID. A second suffix was appended to the URL that would automatically bring the participant to the appropriate survey, since there were many parts to this longitudinal design. A final URL might look like http://some.berkeley.server?uid=XXX&survey=YYY.

This technique worked well enough, but as a cautionary tale, beware the consequences of cut-and-paste. Essentially, this design would make every URL unique, because a given user would only take a given survey once. Research assistants were charged with creating these URLs and mailing them out to participants, but unfortunately RAs did not always perform this task properly, and instead simply cut-and-pasted the URL, thereby making it not unique. As a result, participants took certain surveys multiple times.

Fortunately, for the cases that this happened in, the questions were identical from one survey to the next. The design of the survey required these particular questions to be asked repeatedly over 10 days, so it was possible to recover 100% of the data by inspecting the data journal for its timestamps. By knowing that answers were to fall within 24-hour windows, we re-assigned certain answers to the correct survey.

Although the data journal saved the day, the best practice here is to create software specifically for managing email. This would have removed the responsibility for research assistants to create the proper URL, and it would have assured that a properly generated URL was sent out each time.

"Scales" versus "Pages"

Survey research in general, and social psychological research in particular, makes use of existing scales because they have already been validated. A survey that has many pages might reuse a single scale several times, in different contexts. There are two approaches to constructing the survey: on a per-page basis, or on a per-scale basis.

The advantage to creating scales that are independent from pages is that the scales can then be reused, by simply referencing them from different pages. This can be accomplished by analyzing the various contexts in which a given scale is used, and by generalizing the scale to remove any contextual elements. Such contextual elements must then be provided by the page, so that the scale is rendered in the proper context.

The task of generalizing scales boils down to identifying place-holders for the words that will change, as in the dynamic elements section. For scales that only appear once, this work is unnecessary. For scales that vary too wildly from one context to the next, this work is likely to be more difficult than to simply create multiple copies of the scale.

The advantage to using a per-page basis is that it minimizes the occurrence of dynamic elements, thereby making it possible for the survey to be largely static. The consequence is that the survey can be written in a static format, such as by creating individual HTML pages that do not change. The disadvantage is that very little can be reused, which might increase labor.

Consider the case that a single scale appears 12 times throughout the survey. During development, it is determined that this scale is not implemented correctly. The penalty for using a per-page basis is that this must be corrected in all 12 cases, and the possibility exists that small differences will sneak in during this process.

The major disadvantage to the per-scale basis is the introduction of run-time errors. When items are static, it is possible to review them and determine that they are correct. Then, relying on the fact that it will not change, it will therefore continue to be correct. When items are dynamic, then an unanticipated context might cause these dynamic elements to render in unexpected ways. Such run-time errors can only be uncovered through extensive testing, which can be time-consuming.

Appearance: HTML and CSS

Web-based surveys appear in web browsers, and must therefore be represented using the language that browsers speak: HTML and CSS. As was discussed before, these languages suffer from poor standards compliance by various browser vendors. Therefore, it is important to use the most widely supported components of the languages.

As an example, browsers are able to collect data from users through "forms" and forms can be submitted by either "buttons" or "input submit" elements. Support for the "button" method is less consistent that the "input submit" method, so therefore the "button" method should be avoided.

Generally, the most trivial layout elements should be combined to create the visual presentation that is desired. There is always more than one way to do it, but it is frequently useful to favor the oldest available method.

It is also essential to perform extensive browser testing, because different browser versions from the same vendor may behave differently. It is also not safe to assume that all users will have the newest version of a given browser.

As of the time of this writing (July 2008) my recommendation is to verify your HTML and CSS with the following browsers:

- Internet Explorer 7
- Internet Explorer 6
- Firefox 2.0
- Opera 9.5
- Safari

This will require access to a Windows XP/Vista machine and to an OS X machine, but such access is necessary given that participants will be using a wide variety of platforms and browsers.

Conclusion

Online survey techniques are mature, and will dramatically simplify the data collection process, but the technical challenge is significant. Although it may be possible to rely on volunteer labor (e.g. research assistants) for processing paper surveys, it is not responsible to do the same for online surveys. At this point, too much responsibility is concentrated in too few hands in the case of online surveys, so professional assistance is indicated. This frequently requires a larger up-front financial commitment to the project, and it also means some survey goals must be adapted from their original concept in order to be fully incorporated into an online format. The project will suffer time-delays from unanticipated changes that might not otherwise occur in the case of a paper-format survey. This survey methodology is appropriate for highly developed survey concepts whose designs have already been completed. Finally, YMMV (your mileage may vary).