Distributed Processing Via Email

by Aaron Seidman

Abstract:

The local Association for Computing Machinery (ACM) Chapter has a database of about 4000 names, including members, vendors, speakers, etc. The data is used by the treasurer and several different committees, each of which needs different sets of information. In the past, each data user has had to keep its own version of the database to support its activity, but lack of synchronization and fragmentation of the chapter data created problems. A new, integrated database has been designed, using M and a graphic user interface that can be synchronized via email. It allows each stakeholder to be responsible for the key data on which it depends and uses both an email-transmittable locking mechanism and secondary synchronization to control updates. The user interface is designed to make it easy for a variety of volunteers to use just those parts of the system they need. This combination of database control and easily used graphic interface makes it possible to delegate tasks to many volunteers, each using his or her own computer.

The Problem

The Greater Boston Chapter of the Association for Computing Machinery is an awkward size for database management. It has too much data to be maintained easily by a single volunteer, but it is not large enough to support a professional staff effort. In recent years, paid membership has been in the 800-1000 range, but the database also has expired members and some special categories: institutional members, honorary members and various vendor, professional and other chapter contacts. All together, there are about 4000 names, addresses and ancillary items in the database.

The data is used by different parts of the chapter in different ways. The Membership Committee has primary responsibility for tracking membership expirations, keeping addresses current, and producing mailing labels for chapter publications. The Professional Development Seminar (PDS) Committee conducts six all-day seminars every year and needs to have registration and contact information (such as phone numbers), as well as current addresses in order to manage these events. Members are asked at registration or renewal time if they are available to work on chapter activities and the Volunteer Committee uses this to recruit people for the other committees. The treasurer needs to be able to trace payments and link them to the appropriate activity. (e.g., Sometimes a company will pay for several of its employees to take a seminar and there needs to be a way to connect each of them to the same check.)

Historically, the problem has been addressed in two ways. Initially, each component of the chapter maintained its own database, with only loose coupling to the other repositories. Not surprisingly, this resulted in discrepancies among the different collections and sometimes it was difficult to determine which version of some datum was correct. When these differences involved questions of money it created some awkward situations.

At one point, a member volunteered to collect and maintain all the data in a central database, using a customizable commercial software program. This integrated database was much more useful, but created another set of problems. Clones of the database could be made available when needed by other chapter components, but much of the data entry had to be carried out by the DB volunteer or by someone under his supervision. (Some concurrent updating is permitted by the software, but mergers must be done with binary copies of the database.) The time required to maintain the database made it difficult to recruit a replacement for the volunteer and left the chapter highly dependent on a single point of failure. (Although the database has been conscientiously backed up, the software program that maintains it is complex and has been highly customized; only one person really understands how to use it.) It also turns out that the database is of a proprietary design and on those occasions when data gets corrupted, the whole database has to be sent off to the software company to be fixed.

We have also found that we did not completely understand all the data we did have, because the software does not maintain a data dictionary (or at least not one that the user can access). This is important in a volunteer organization with sometimes unpredictable personnel turnover, where the outgoing officer/chair/registrar/etc. may be about to disappear from the area and forget to tell the incoming user about key data structures.

Goals

The chapter needs a database that is accessible by a number of authorized individuals, can be updated by those who collect and handle the data, and that does not require extensive training of those individuals who need to use it. It has to be able to run on individual PCs (Macs would be a plus, but not essential).

We started with the idea that we would like to retain the integrated database, but make it possible for maintenance to be more of a collective enterprise. One way to do this would have been to put everything on a central server with password protection and allow all appropriate individuals to dial in or connect via telnet or ftp to update or retrieve the data. Unfortunately, this is not currently an option for the chapter, although it may be in the future.

The database, although it appears large to those trying to maintain it on a volunteer basis, is actually fairly small in terms of the number of bytes it occupies on disk. In fact, in ASCII form, it is feasible to email it. Typically, this is the way in which we transmit labels to our mailer. Size, therefore, would not prevent us from using email as the distribution medium. (While moving the database via email constrains scalability, we anticipate that well before we hit practical limits we will have shifted to some kind of central server system.)

We would like to have the treasurer enter payment information, have the PDS registrar enter PDS-related data, have the Volunteer Committee update preferences and email addresses for activists, and the Membership Committee handle renewals and changes of address. The problem is, of course, not how to let them modify the database, but how to synchronize the multiple copies of the database so that everyone has the appropriate changes that have been made by others.

Finally, we need a user interface that is as intuitive as possible and at most requires only a few minutes of instruction. We want to be able to maximize the number of member volunteers who can help maintain the database and, if necessary, fill in for (and be successors to) lead maintainers.

Solutions

We needed to find suitable technology, devise a suitable user interface, and solve the synchronization problem. (We also designed it with a data dictionary, but since there is nothing particularly innovative about that, we do not discuss it here. For those not familiar with this technique, see the appendix for an example.)

Technology

There is no theoretical reason why this project could not be implemented in other types of database systems, but for ease of maintenance and storage efficiency, we elected to use M as the database engine. It would be nice if databases never became corrupt, but with M it is easy to fix application-level corruptions and possible for an appropriately knowledgeable programmer to repair low level errors without having to go back to the vendor.

We considered using a text-only user interface, but quickly rejected it. First, there was a question of acceptance. Today, we are so used to graphic interfaces that non-GUIs are immediately discounted and interpreted as a signal that one is dealing with an obsolete system. It was important to the chapter that members feel they are working with current technology. Second, a well-designed GUI can convey more information more effectively than a text-only system. (A poorly designed GUI may actually be worse, but that is another discussion...) We decided that because there were several M systems that already work with Visual Basic and because we have some experience with it, it made sense to use that for the graphics. Another consideration is that one can use essentially the same VB system on Windows 3.1, Windows 95, and Windows NT systems, thereby making it easy to support multiple platforms.

The system is currently under development, using Micronetics Workstation for Windows (MWW). Two factors influenced this decision. One, MWW is being developed to run on the same platforms that VB already operates on, Win3.1, Win95 and NT. Two, we can produce as many royalty-free .EXE files as we need, without having to get an M license for each machine on which we run the system.

The system could have been built with other interface tools, such as Delphi, or with Visual M; these choices were more practical than theoretical.

Graphic User Interface

One of the ways of making the interface easy to use is to isolate the various functions it needs to perform and create individual interfaces to each function. Thus, we will have a screen for new member entry, another for entering dues payments for an existing member, a screen for entering volunteer information, etc. Potentially, we can use this technique to limit certain kinds of updates to particular functions and thereby simplify the synchronization. For instance, if the copy kept by the treasurer is the only one that allows payment entries, then whenever we merge copies of the database, we simply let the treasurer's payment data overwrite any other. This turns out, however, to take care of only a fraction of the problem.

In practice, most users of the system have to access a wide range of data and there is considerable overlap in what different committees would need to update. This means that limiting the interface would not be a way of solving the synchronization problem. This does not mean that it is not a good idea to have specialized interfaces for each of the functions, but that we should not use human-computer interface design to solve a database problem.

In the end, we decided to go with the customized-by-function approach, subject to revision based on user feedback. We hope to get additional feedback from some of the local ACM SIGs; at least one of the membership chairs has agreed to act as a beta tester for the software. Although we are designing it so that we can impose access restrictions on individual copies of the software, we will probably not activate that feature because we know the 6-8 specific individuals who will be using it. (We are just using a standard control system with a global containing passwords and access codes. When activated it requires the user to give a password and then only shows the interface screen(s) the user is authorized to work on or view.

Synchronization

The most important problem—and probably the most interesting for this forum—is how to keep the multiple copies of the database in some reasonable harmony with one another.

With a centralized server, one would just use standard locking schemes and users would operate in a real time mode. We had already determined that this was not feasible at this time.

An alternative would be a centralized depository, in which a user checks out a copy of the database, leaving a flag that signals subsequent users that it is available for read-only uses until checked back in. By locking the entire DB it would significantly inconvenience other users and slow down the process by which we keep our information current.

There is another potential problem as well. While we trust our users not to do deliberate damage to the database, experience teaches us that volunteers can get distracted by real job issues, family matters, and other things. We could end up with a queue of people unable to enter updates because the current holder of the update token has not yet checked in the revised DB. This is reminiscent of DSM systems in which a system manager has killed a process with an operating system command rather than using the appropriate DSM utility and all the other DSM jobs would grind to a halt. The only solution was a system reboot.

In a standard, shared database, the most recent entry is the current one. When merging two versions of a database, we normally wish to take the newest entry and discard older ones. Depending on the nature of the system and the desired granularity, the system would lock the appropriate record or record group, execute the update, and release the lock. (The details may vary depending on how busy a system this is, whether it is using transaction processing, etc.) The important thing is that an update to one field not undo updates to another field. The classic textbook illustration is where, in the absence of locks, two users get the same copy of a record, modify different fields, and file replacement records. The last to refile preserves his changes and wipes out the other change.

This kind of consideration made it clear to us that we would require very fine granularity—down to individual fields—and a way of determining, in any database merger, which field was the most recent. The obvious solution was to time-stamp each field in the database. Then, when a user has performed a series of updates, she can have the system write an ASCII file with all the data, and email this to all other users. They, in turn, can read in her data, and the system will compare the dates on a field-by-field basis, filing the one, thereby insuring that everyone is now operating on the latest version.

There are two problems: The first is that adding a time-stamp component to each field significantly increases the size of the database. While this may not be critical for the M global version, given the wide availability of large PC disks, it could expand the ASCII dump significantly, making it more difficult to distribute by email. A second problem is what to do in the case that two different field stamps have the same value. Since most fields will, in fact, have identical time-stamps, we would have to do identity checks on the value of every field in order to detect such events.

We decided that the simplest solution to both problems was to use the following form of data compression:

The program has a provision for initializing the database, providing a base time-stamp ($H) for the whole thing. Then, all we need do is mark the exceptions. Whenever a field is modified, the record is marked and a time-stamp for the field is added to the record. Similarly, new records are also marked. During a merge, unmarked records from the incoming database are ignored, since we can assume that they already exist in the resident database, either in the same form or in a later, updated form. Whenever we encounter a marked record, we look for time-stamped fields and see if they are later than the comparable fields in the resident database (which is likely to be the initialization time for most fields). Incidentally, this means that deletions are not actually removed from the database, but simply marked as deleted.

In the case of identical time-stamps on a field, the system compares the value of the two instances of the field and, if they are different, reports to the user. (Lest one think that

this is an improbable, if possible, occurrence, consider the following scenario. Copy A is updated, then merged with copies B and C. Next, B is modified and merged with A and C. During this second merger, all the databases have marked fields with identical time-stamps from the first merger.) This, of course, introduces another complication, namely that after a few mergers, the number of compares on each subsequent merger increases significantly, thereby slowing the process.

We have provided for one simple solution, but if that does not work well, we will go to a more sophisticated approach. The simple solution is to reinitialize the base time-stamp periodically. Given the size of the database, and the processing power of contemporary PCs, it is not clear that the slowdowns from multiple compares will be that noticeable. If it is, and given the relatively small number of users, it should be fairly easy to arrange for all copies to be synchronized and then have all users run an initialization (or alternatively, send a new version with a new base time-stamp; if the incoming base time-stamp is later than the one in the resident version, the incoming data overwrites the resident data).

Another possibility is to modify the marking system to indicate that a merger has occurred and ignore records from the same merger in other copies. Although more interesting from a theoretical standpoint, and probably important were we dealing with a larger user base, we have opted—in the interest of time and programming simplicity—not to implement such an approach at this time.

There is one final issue, and that is how we avoid duplicate keys if we allow the entry of new records in more than one copy of the database. An important requirement for a system like this is that we have a unique identifier for each person in the database. Because names cannot be guaranteed to be unique, we assign each member a serial number. However, if we allow more than one person to be updating the database simultaneously, how do we make sure we don't have duplicate numbers?

One way is simply to assign blocks of keys—our five-digit serial number has no significance other than being a unique identifier. We could simply make sure that each copy of the data base generates a different set of numbers. For instance, one copy could generate numbers in the 10000-19999 range, another could be limited to 20000-29999, etc.

A second option is to allow only one copy at a time to be used for new record entry. In this approach, a "master copy" carries a token that enables new records to be entered; in all other copies, updates can be made only to existing records. When a user dumps the data to an ASCII file, one of the options is to dump the token as well. The software marks the token as absent and controls entries accordingly, until it is reloaded. The token can be emailed, along with the data file, to the next person doing data entry. The person receiving the token (some kind of number, encoding information such as date and latest serial number) can load it and operate that copy as the "master." (Note that the token must be dumped simultaneously but separately from the data; we only want the token to go with one copy of the DB. One of the operator options at the time of a data dump would be a token dump.)

A third alternative is to allow the creation of files of new record data—without keys—that can be emailed to the holder of the master token, for automated batch entry. The software would have to be told, either by operator action or by a marker at the head of the file, that the data represents new information, and then it would simply assign serial numbers as it loads the new member information. This latter has the advantage that any of a variety of software entry packages can be used for this purpose, including editors, word processors, and spreadsheets, as well as our system.

Theoretically, all of these approaches are compatible, and we have not yet decided whether we will use just one or a combination. We are currently leaning toward the first and third options.

Discussion

One may ask why we are emailing the entire database instead of just those items that have changed; that would, after all, be a logical extension of the data compression design. The answer is that we may go to that kind of exchange in the future, but there are several reasons why it is not in our first version of the software.

This project is interesting from a design standpoint, but there are some important engineering considerations. It is, after all, a real system, designed to do a real job, so at a certain point, we had to decide what we could implement in a reasonable amount of time. Those of us with experience in the commercial marketplace are familiar with the "one-plus" syndrome in which people keep coming up with "just one more" added feature for a new product. By the time all of these features have been added, the release date has slipped by a considerable amount. In addition, some of us have learned the hard way that incremental development is more likely to lead to reliable software. This is especially true with something new because it is very difficult to be confident that one has thought of every contingency. We also felt that it was safer to have multiple instances of database copies (easily identifiable by date and creator in a header record), than to rely on a single instance of each copy. It is, in effect, a redundant backup policy.

Appendix:

The data dictionary is simply a global holding pointers to the global locations of the various data elements, along with basic information about the element's characteristics. For instance, the entry for a phone number might look like this:

```
^DD("PHONE","GBL")="^REC(ID,[PH],INDX);1"
^DD("PHONE","TYPE")="^REC(ID,[PH],INDX);2;
2;W;H"
^DD("PHONE","FMT")="10N;(3N) 3N-4N;1,3,617"
```

This tells the system that the data item "PHONE" will be found in the ^REC global, which has the master key "ID," the literal, "PH" and is an indefinitely repeating field, with the individual cases marked by "INDX"; the phone number is the first piece. The phone "TYPE" is the second piece of the data, and it may assume two values if present, "W" and "H." It is stored internally as a 10-digit number in the form (nnn) nnn-nnnn and the default value for chars 1-3 is 617.

Using a data dictionary means that one can modify entry forms and reports without having to worry about the details of storing and retrieving data elements each time; the system keeps track of where to file information. It can have some impact on performance, since storage and retrieval operations have to be written to go through the dictionary. For high-performance systems there are ways of optimizing but we do not think that will be necessary for our purposes.

Another benefit of a data dictionary is that it makes it easy to get a listing of all data elements stored in the system. One simply has to have a routine that reads the ^DD global and formats it appropriately. **M**

Aaron Seidman is a principal of Imaginative Illustrations, a firm specializing in Web page construction and design for business and nonprofit organizations. He can be reached at 617-232-2509. Email: aaron@imaginillus.com

---