Implementing the ReadyCall service - Buddy lists for phones.

Greg Kochanski, Daniel Lieuwen

May 14, 2002

Abstract

What if you didn't have to dial the phone? What if people called you when you wanted to hear from them? The ReadyCall service does the trick: it connects people who mutually want to talk to each other.

We discuss the implementation of the database and server that arranges the calls. A single PC can cover a large corporation, and even a system covering the entire United States is technically feasible.

1 Introduction

ReadyCall [2, 1, 3] is a telephone service similar to AOL Buddy Lists: Each person has a list of people they would like to hear from, and when two people with matching lists are both available, the system calls them both and connects them.

In the early 1990s, when the service was imagined, it wasn't practical to know when people were available, mostly because there was no easy way to put an extra button (to allow user to indicate their availability) and light (to allow the user to see their current availability) on a telephone. However, with IP networks starting to go to telephones, and cell phones with screens and buttons, the service is now easy to roll out.

This TM discusses how to implement the central server that matches people, makes the calls, and connects the pairs of people.

2 Overall Implementation

In its simplest form, ReadyCall requires a pair of phones, each with a switch to mark when the owner is *available* and willing to receive calls. The implementation we describe here is more sophisticated than this basic form, making a distinction between persons and phones—thus allowing people to share phones without confusion. Each person also has a list of people that they want to hear from (and some way to edit the list, of course). So, if there are two people subscribing to the service who have mutually indicated that they wish to hear from

the other, and if both are home, then the system calls them both and connects them together.

The system divides nicely into three parts: a database, a "matching server" that computes which people should be connected together, and a dialog system that makes the phone calls.

The size of the dialog system scales with the number of calls being made, and can be built by simply replicating the hardware required to make a single call. It is loosely coupled to the central database/server: a call is initiated by sending a single message to the dialog system which just tells which people to call and phone numbers.

When a call terminates, its success or failure is sent back to the central database/server, to build a record of who is available when, which can be used to improve the user experience. For instance, the system may learn that certain users tend to leave their *available* switch on, even after they are no longer available. For those people, the central server may automatically turn off *available* after, say, 30 minutes.

3 Databases and Matching

We will call the three database tables involved the *available*, *wants-connect*, and *identity* databases.

To implement ReadyCall, one needs to solve *available*(A) & *available*(B) & *wants-connect*(A, B) & *wants-connect*(B, A), and track when new (A, B) pairs are added to the solution.

3.1 Database Tables

The *identity*(*external-id*, *internal-id*) table maps a "globally" unique[1] external identifier (*e.g.,* social security number, name/address, AOL screen name, email address) to an internal unique system identifier coded as an integer.

The *wants-connect* (*caller*, *callee*, *state*) table contains records indicating that a given *caller* wants to contact a *callee* (both represented using the integer unique identifiers from the *identity* table). It also contains the time of the last connection through the service, and some information that says when another call might be appropriate (*e.g.,* call once per month, or call after 8 pm).

The *available* (*callee*, *phone*, *reachable*, *heuristic*) table contains the system identifier of the the callee who wishes to use the service, the phone number where *callee* can be reached, a boolean indicating whether or not the callee wishes to use the service, and maybe some information for heuristics that decide the odds that *callee* will actually answer the phone. Also, it could contain information used heuristically to decide when to turn off "available".

[1] Globally as far as this service is concerned, so could be a person's corporate id for a corporate site.

The algorithm operates by monitoring changes in states. It receives a notification from a web, IVR or other interface that a user, U, has changed state from "unavailable" to "available" (or the reverse).

In both cases, the *available* table is updated, but upon entry to the "available" state, the system also checks to see if any calls should be made. To do this, it iterates through U's list *wants-connect*(U, b, state), checks the extra state information, to see if a call is appropriate, and then checks to see if *available*(b, *, *, *) is true. If so, it checks to see if the tuple *wants-connect* (b, U, *) exists.

If so, it calls both parties separately with a dialog system, and asks them if they will accept a call from the other. Finally, if they agree, it connects them.

We can calculate the database resources needed fairly simply. Assume that the system has enrolled N users. Each *identity* record takes perhaps 8 bytes. Each *available* record takes perhaps 16 bytes. If we assume that each user keeps M people on their *wants-connect* lists, and each entry is perhaps 16 bytes long,[2] then the total storage is (24 + 16*M)*N bytes.

3.2 Memory Footprint

For M=100, the total storage for a system covering all of the USA (100 million people, which is comparable to the total number of cell phone users) is about 200 GBytes. This is clearly a practical size. Smaller, corporation-sized systems with N=100,000 could easily fit in RAM memory of a generic PC.

In a large system, the appropriate design is to keep the *identity* and *available* tables in RAM, and leave the larger *wants-connect* table on disk. In that case, RAM requirements are 24*N bytes, so even a nationwide system would fit into 2–3 GB of RAM, and could be accessed quickly. This design limits disk accesses, because the information in the *available* table can eliminate many possibilities.

So, even large systems will fit into the memory of a normal PC.

3.3 CPU speed

What about CPU usage? We will assume that an average user gets/makes 3 calls per day via the system (business users might get/make 10 or more, home users perhaps one), and that users flip their *available* switch also about three times daily.

For a corporate system supporting 100,000 users, the database then needs to handle 300,000 *available* updates per day, or a peak rate of 30 per second (assuming a 10:1 crest factor). This is well within the capabilities of a single-processor PC, even if it weren't true that the entire problem fits in memory. (Actually making all those phone calls will require more than one PC for the dialog system, of course.)

A nation-wide system is beyond the capabilities of a single PC, as it must process 30,000 *available* updates per second, each of which requires an average

[2] Encode each person with an integer identifier, encode time of last connection in an integer, and encode information about appropriate time to call in an integer.

of 3 disk accesses. However the problem does split cleanly: a processor could handle the computations for a few million people. The database is likewise segmentable by region (or by any arbitrary division). We note that perfect synchronization of the sections of the database is not important: modest (1 second) delays in updates will simply make the system behave as if the *available* switch were flipped slightly earlier or later. The system might miss an opportunity to make a call if A flips *available* on just before B flips it off, but users would be unlikely to notice, and the call would be made later.

A spurious call might be generated in the reverse situation, if A flips off just before B flips on. Part of the database might see those events in the opposite order, and a call might thereby be initiated. Here, user A will be called shortly after he/she flips off *available*, which he or she might consider odd, but it should be a rare event. With one second of update skew, a user might expect a spurious call once a year or so. Further, the call will only occur within a second or so of switching *available* off, so the user is presumably still at the telephone and still awake. These spurious calls are indistinguishable from legitimate calls that are delayed by processing in the dialog system; we doubt that they will disturb users.

4 Enhancements

A refinement then comes to mind: contexts. For instance, a person in "work" context wants to chat with colleagues but not family/friends, and vice versa in his/her "home" context. A person should be able to have several contexts and be able to choose which of them they want: to be *available* to some groups of people and not to others. This will eat up more disk space for the nationwide case, but should not otherwise affect the service. Within a company, such a feature is optional, useful if you want to focus on one project for a day, and delay work on other projects with other people. As a consumer service, though, we think it would be very valuable.

One could consider the simple *available* switch we described in the first sections as a degenerate case of a single context. Of course, even if you have many contexts, you could be *available* to several of them at once, or even all of them.

Adding contexts to the system requires modest increases in storage space and computation. For instance, the *wants-connect* table then needs an extra field to show in what contexts you're willing to take the call. This could be a single bit-mapped integer, allowing 32 contexts and adding only four bytes to each entry. Likewise, the *available* table would also have a bit-mapped integer added to show in which contexts you are *available*.

Finally, the system would need an extra table containing the names of the contexts. This table would be large, especially if the context names are different for every user, but it is accessed relatively rarely: and can be stored on disk, as it is not used as part of the main computation.

4.1 Reducing Disk Accesses

A good way to reduce disk accesses is to put a clustered (hash) index on the *wants-connect* table on the *caller* field and to add a "mutual interest" field to the table. Then, upon a new user U becoming available, U's *identity*, Uid, is looked up and a new *available* record is added in memory. Then, a lookup is performed on the *wants-connect* table using the clustered index lookup for *caller*=Uid. If the record is marked as mutual interest, then do a in-memory lookup to see if the other party is available. Otherwise, look at next record. Only one or two database page reads will be required to find who a person wishes to contact if a hash index is used and two to four if a B-tree index is used (assuming that the upper levels of the tree are cached). By including a pre-computed "mutual interest" bit in the *wants-connect* table, these are the only database page reads required for an *available* update.

Maintaining the "mutual interest" bit is trivial. Whenever U adds a new *wants-connect* record R1, the *callee* field of that record is first extracted. Then, the clustered index is consulted to find all the *wants-connect* records for that person. If the record R2=(*callee*, *caller*, *) is found, then both R1 and R2 should have the "mutual interest" flag set and be written to disk. Otherwise, R1's "mutual interest" flag should be set to false before writing it to disk. Thus, it requires two index lookups for R1 and R2 and up to two tuple writes and one index write in the normal case (one or more index updates are possible in rare cases).

Deletes are similar. When deleting a *wants-connect* record, no other work is needed unless the "mutual interest" flag is set. If it is set, two index lookups and two writes are needed (given that coalescing of pages is generally not implemented in indexes, exactly one index update will be expected).

While new customers may add a large number, say 30 entries, on their first day, typically a user will average some small number (less than five, averaging perhaps one or two) of add/deletes of *wants-connect* records). Assuming two adds/deletes per user implies 200,000,000 of these updates/day. However, these updates do not need to be made in real time. Furthermore, they will tend to be clustered–a person will tend to make multiple changes in one day and none for several days after that. Taking advantage of this by grouping adds/deletes from multiple updates into a transaction can help further.

The simplest way to do that is to accumulate all the updates that come in for say five minutes and apply them in one large transaction. This will mean that most of the time, there will be no lock contention between processing *available* and *wants-connect* events. When it is time to add *wants-connect* records, the system can refuse to process any *available* events until the *wants-connect* records are done being added if transactional consistency is required. However, it is probably acceptable to avoid conflicts entirely by running at a reduced consistency level for the *available* events. At worst, they will receive information that includes some combination of seeing only a subset of the new additions and seeing a subset of the items specified as needing deletion.

5 Conclusions

A single PC can run a ReadyCall system inside even a large corporation. A nationwide system would require careful design of a multiprocessor system, but the problem readily splits both at the computational and database levels, so that ReadyCall can be scaled up to cover very large populations.

References

[1] C. Fernandez, A. R. Just, and G. P. Kochanski. Telecommunications system for dynamically selecting conversation topics having and automatic call-back feature. U.S. Patent 5,596,634, January 1997.

[2] G. P. Kochanski. The basic readycall service. Technal Memorandum 11119-920828-51TM, AT&T Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974, 1992. Now Lucent Technologies.

[3] G. P. Kochanski, C. Fernandez, and A. Just. The dynamic chat line – talk about anything, anytime, anywhere. Technal Memorandum 11177-940830-12TM, AT&T Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974, 1994. Now Lucent Technologies.