Centrally managing an ISP with NetBSD and PostgreSQL

by

D'Arcy J.M. Cain, Vybe Networks Inc.

Abstract

Vybe Networks operates as a comprehensive Internet service provider Our goal is to create a system that can react to changing markets without losing control of the system usage. We also want to be lean and agile. The system should support all of these goals and be simple to administer day to day.

First, a little about me and how I arrived at the requirements that I address in this paper. Then I will discuss those requirements and, finally, show how I implemented a system to meet them.

I am currently co-owner and CTO of Vybe Networks Inc., an agile software development and network management company that absorbed my previous company's operations, Vex Consulting, and Jim Adams' company Vibe Online. Previously I was Vice President of Greenlight Communications Inc. following a stint as Vice President of Internet Canada Corporation. and their sister company Internex Online. I was also Chief Technology Officer for Givex Canada Corporation.

Before that I was a director of Planix, Inc., a company doing Unix and Networking system administration and consulting.

I have also worked for Monarch Marking Systems, a Pitney Bowes company, BDH Computer Systems, an early microcomputer systems integration and sales company as well as running my own consulting service as D'Arcy Cain Consulting.

I program in a variety of languages specializing in 'C' and Python. I have also programmed in assembler for the PIC Processor, 8080, Z80 and iAPX family (80x86), Bourne shell, Tcl, Tcl/Tk, Python, SQL and Progress 4GL and have some experience with BASIC, PostScript, awk, Perl and others. I am a contributor to the Usenet comp.lang.c FAQ as well as the comp.os.msdos.programmer FAQ and have been published in a compilation called "Software Solutions in C" edited by Dale Schumacher (ISBN 0-120632360-7), an honour I share with people such as Douglas A. Gwyn, Henry Spencer and others. I have done a little Winsock programming although I am not primarily a Windows programmer.

I am involved with the NetBSD and PostgreSQL projects and I am the principal maintainer of PyGreSQL, a Python to PostgreSQL interface. In addition I have written and distributed many programs under an open source license.

In the late eighties I took over the operation of a brand new ISP that was suffering from serious administrative issues. From the start I could see that the number one problem from both a technical and business standpoint was that systems were not co-ordinated. Users were added to the service and, usually, were also added to the billing system. It was possible for users to be on the system without any knowledge by the administrative system. It wasn't even necessarily done on purpose but if someone realized that they were accidentally given a free ride you couldn't always expect them to let you know.

Over the years the problems only got worse as services that were profitable at the start soon were being delivered for much less and often for free. This meant that we had to keep adding services to keep relevant. Hosting, email, domain registration, phone service, database management, backup and specialty services were being added all the time. With markets pushing down prices we had to manage more and more services with fewer staff relative to the work.

As a specialty shop we also needed to react quickly to client needs. We had to be able to quickly add services that the bigger ISPs just couldn't do.

After a while we took over another ISP who had a system that was better at tracking all the users but was maintained as a collection of Perl scripts and flat files. The admins were very good but even there the chief programmer was spending too much time dealing with individual user issues. The system could not be run without highly technical people interfacing with the users. The system needed to be much more user friendly.

The requirements that grew out of this were as follows.

Nothing should happen in the system that is not controlled by the content in the billing system. No ad-hoc services. Everything that is used has to be tracked and billed although there may be products that are not charged for. It should always be possible to query the database to find details of anyone's use of the system.

While most things charged will be products in our catalogue we also need to have the ability to create unstructured billing for things that don't quite fit. This has to handle recurring billing (e.g. monthly management fee) or one time (e.g. modem purchase or consulting).

The system should be modular. The central module would be the billing system with ISP management module hooked into it. It should feed the A/R and A/P which in turn feed the G/L. There should be a simple way to break this chain anywhere and feed a different system if desired.

Even though the modules are distinct they should be presented in a consistent way to the user. The administration system should be intuitive and friendly. Errors should be caught with good feedback. The system should be self monitoring with reliable alerts. Sending an email to alert the administrators that email is down won't work.

The system should be configurable. The enterprise using the system should be able to make local configuration changes without having to change code. Presentation should be separate so that the enterprise can have an interface that matches their other systems.

The system should be able to create new products by defining the characteristics such as price, term, description, and other details. The end user should have control over some features as well.

The system needs to be robust enough that one server (e.g. the central admin server) isn’t so critical that the entire system goes down if it does such as a reboot or hardware upgrade. This means a pull protocol where each server (mail, web, VoIP, etc.) will connect to the central server from time to time to check for changes and build local configurations and databases to run its own service or services.

All of this should be running on reliable, open source software on commodity servers.

Since I had previously done a lot of work with relational databases I felt that a properly designed database using proper ER techniques was the first step. I worked out a basic design for the various objects that had to be managed and made sure that the relationships protected me against major errors such as users with no billing information.

I needed to run this database on a reliable, enterprise level engine. There were a number of commercial ones that I checked but they were either too expensive or they didn’t seem to be sturdy enough. I looked at MySQL because of its popularity but it had too many issues especially with transactional management.

I also looked at Ingres which was a research project run by Michael Stonebraker at the University of California, Berkeley which had promise but was no longer being supported. It had been moved into a commercial product. In 1985 Dr. Stonebraker returned to Berkeley and started working on a new database continuing the work started with the Ingres database engine. Since it was post Ingres he named it “Postgres”.

In 1994 two students, Andrew Yu and Jolly Chen, took over the project which by then was under an MIT style license. In 1995 it was released under the more liberal BSD type license and renamed Postgre95, a jab at Windows95. This version also changed the query language from the Ingres inspired PostQUEL to SQL. Later the project was rename PostgreSQL to reflect its support for SQL.

I was already a huge fan of NetBSD so that is what I used as the operating system. This was years before I became a developer. While I like all of the BSDs in general I have always found the NetBSD code base the cleanest of all. The developer team while smaller than other versions of BSD tended to be anal about doing things the correct way rather than just getting releases out the door.

Now that I had a solid database engine running on a solid operating system I looked for a language to write the code in. I had previously built an accounting system in a relational database engine called “Progress RDBMS” but the language was not suitable outside of that database. I needed a general purpose language that could access arbitrary databases and PostgreSQL in particular.

I have long been a C programmer and considered it for my language of choice for this project but I felt that it was not agile enough to be able to deal with constantly changing requirements. I looked at Tcl for a while and even started writing code but then I discovered Python.

Python started out as a hobby language by Guido van Rossum in 1989. By the time I discovered Python it was a stable product with lots of support. It was object oriented but allowed for procedural programming as well. I pretty much love everything about this language, even the controversial indentation.

The final piece of the puzzle was an adapter to connect Python to PostgreSQL. I found a project called “PyGres95” by Pascal Andre. The code was not being maintained at the time and needed some work so, with Andre’s blessing, I took over this project and renamed it “PyGreSQL” to keep in step with the PostgreSQL project.

Finally I had all the pieces and was ready to put everything together. Currently I am using many open source components including these:

- NetBSD
- PostgreSQL
- PyGreSQL
- Apache
- Asterisk
- Postfix
- Dovecot
- Subversion
- Rsync

The core of the system is the management of all the services in the database. This is broken down into a number of major modules. The billing module is the main part of the system which I will describe in more detail a little further down. Connected to that module is the ISP module which I will also expound on at length.

The billing module connects to the receivables and payables modules which, in turn, connect to the general ledger. All of these modules are mainly independent and transfer information through simple import tables. The modules above read information from these import tables making this a "pull" system rather than a "push" one. That allows for an enterprise to replace modules if desired. For example, they may wish to use their own accounting package to handle everything above the billing system. They can do that by simply writing a program that reads the arimport table and writing to their own receivables package. Alternatively they could use our receivables system but do the same as above to import the data into their own G/L package. The only thing they need to do beyond that is keep some tables in sync such as the chart of accounts if using their own G/L or the client table if using their own receivables. Simple scripts can be used to do that as long as they choose one system to be the master.

The billing module can be further subdivdied into a few components. This includes clients, services, users, ad-hoc billing, invoicing, invoice payments, commissions and others.

The client table is the place where we start. It tracks all the detail about the user such as the name, address, billing email and and other billing details. This includes usual information about credit limits, payment status, etc. With few exceptions this is a standard client table.

Under the client table we have service groups, recurring billing, invoices and invoice transactions. Service groups is where we define the products we offer such as web hosting, domain registration, virtual Internet services, telephone services, connectivity, etc. Services are defined in a table that defines term (usually 1 or 12 month), billing period (for extra charges such as disk space overage and telephone minutes), base price, etc. It also defines the data that we need to fill in. Here's an example.

We have a product called "NetTask Manager" which is an online task management system. We create a service for it and name it "NETTASK". It has a price for the base and a price for users above the included number. This service can be attached to a service group that is attached to a client.

There are a few other tables that also define what our clients use. Since they are all connected to the service group which is in turn connected to the client no service can exist for a non-existent client.

The ISP module is closely related to the billing module but is still a separate entity. User accounts are attached to a service group. While the service group defines whether a user has a web page or email it’s the user account that defines the details.

The significant aspect of the system is how services are provided. Rather than having all the services pushed out each service decides what it needs from the database and pulls it down. So, for example, the email is managed by having the email server update the password file based on active users in the system and then updating the various configuration files for Postfix and Dovecot. If the central server ever goes down for any reason such as maintenance the existing files continue to work until the admin server comes back up.

Administrators have the ability to turn off individual accounts which will stop any services connected to them. They can also turn off a service group which will effectively turn off any users attached to it. In the same way, turning off a client turns off all of the service groups attached to it. In this way no one can receive services unless the billing system is aware and charging them.

Services become easy to create because the smarts are moved down to the system providing them. Here is a case study of one such service.

We decided to add phone service. To do this we first created an entry in the system to define and price it. Once that was done we then installed Asterisk on a server and created scripts that would generate the Asterisk configuration files. On a regular basis the script checks if the relevant tables for this service have changed and if so then the configuration files are re-created and the Asterisk server restarted.

Since we charge above a specified value for usage we need to monitor call times. We do this by having Asterisk create call records which we update from a cron job. We don’t deal with counting minutes at that point. We simply work out the price of every call as if it was going to be billed. The billing system uses common code to discard charges for the minutes that are included in the product and charges for the rest.

The same thing applies to all of our services. The individual systems count and measure usage storing the information in the database. The billing process handles actual billing if required.

Not everything needs to be measured outside the system. Once a service is created we can count it directly. Also, user accounts are defined within the system so mailboxes can also be counted directly.

Now that all the pieces are in place we need to add administrative control. We do this through a secure web site with password control. The first step is creating all of the account types and services. This is the harder part of the process because we try to add as much flexibility as we can. We create basic account types such as "Basic Web", "Email Only", "Virtual Domain", etc. For each account we specify the basic data such as name, base price, whether it is a virtual domain account, whether it has a web site and other attributes.

We also define services that can be attached to an account. These include ADSL, hardware rental, mailing lists, PBX, VoIP, zone file, etc. Not everything need to be billed. For example, the last item, zone file, is something we offer for free if a client has special needs not covered by our automated zone file creation but the price is $0. However, by having it in the database as a service we have a place to store the custom zone and making it dependent on the status of the client.

Each service has an included count, a price for overage and a field to specify the units. Most of our accounts have 30 included mailboxes with a price for additional mailboxes.

There are special groups that are used as defaults for the others allowing us to specify common pricing in one place. When a new account type is created it effectively includes all of those items but can be replaced by a specific one for that account type.

The rest is easy. We create clients, service groups and users using simple screens. We select account types and add services such ADSL, etc. The various systems notice these changes to the database and update their local files accordingly. The users can now log in and use the service.

Besides pulling the required data, the system also counts resources and updates the various services and other tables. There are many different ways to do that. Here are a few examples.

Disk space is complicated by the fact that space is used on multiple servers for different purposes. Although we have one price for space overages, others might want to charge different prices for different types of storage. To handle that we have different scripts to gather usage information. The mail server and web server simply track the usage in the user home directories and update the database for the respective storage types. Later, another process gathers all the individual usages and stores it in the disk usage service for that account.

Phone minutes are collected from the CDR table described above. The script accumulates the un-billed amounts and stores the result. The script is simplified by the fact that included minutes are handled in the actual billing later. For now we simply store the raw minutes.

Sometimes we can't count things directly. For example, in our case we do not have direct accesss to our DSL provider's system so we have to go by the monthly invoice detailing each user's charges. For this we have to check it manually. Once a month we have someone go through the invoice making sure that everything that we are being charged for is on the client's service group. Of course, this means that we could theoretically be charging for something that is not being provided but we trust that the client will complain loudly about that.

The next step in the process is to run a script on the billing machine to check all of the services for a client and add a billing transaction. Since we have specified all of the services in a consistent way we don't need to modify our script every time we add a service. As long as it is correctly defined we can let the billing script do its thing.

At the same time we check for recurring billing. As described above, these are things that don't quite fit the model but we do not want to have to add them manually every month. Sometimes we find ourselves with many similar recurring billings and we will find a way to turn it into a service.

Next, we run a script that gathers an un-invoiced transactions and creates an invoice that is sent to the client (usually by email) and creates records in the A/R import table.

There are also a modules for accounts payable and general ledger but those are beyond the scope of this paper. There is also a credit card payment module that is used by the billing system.

That's the basic system. Everything in our system that we charge for, and many that aren't, is tracked in the system and can be queried by anyone with administrative access. Clients can also query some information relating to their own usage of the system.

More information about us can be obtained by contacting us:

- Vybe Networks Inc.
- #203-100 Queen Street East
- Toronto, Ontario, Canada
- M5C 1S6
- info@VybeNetworks.com