APPENDIX C

# ISPs Providing mod_perl Services

This appendix proposes a few techniques for deploying mod_perl on ISP machines. Therefore, it's mostly relevant to ISP technical teams and ISP users who need to convince their providers to provide them with mod_perl services.

There are at least four different scenarios for deploying mod_perl-enabled Apache servers that ISPs may consider:

- Users sharing a single web server
- Users sharing a single machine
- Giving each user a separate machine
- Giving each user a virtual machine

This appendix covers each of those scenarios.

## Users Sharing a Single Web Server

An ISP cannot let users run their code under mod_perl on the main server. There are many reasons for this. Here are just a few to consider:

*Memory usage*
: One user may deprive other users of memory. A careless user's code might leak memory due to sloppy programming. A user may use a lot of memory simply by loading a lot of modules. If one user's service is very popular and gets a lot of traffic, there will be more Apache children running for that service, so it's possible for that user to unintentionally consume most of the available memory even if she has a very small, well-written code base with no memory leaks.

*Other resources*
: It's not only memory that is shared between all users. Other important resources, such as CPU, the number of open files, the total number of processes (currently there is no easy way to control the number of mod_perl processes dedicated to each user), and process priority are all shared as well. Intentionally

or not, users may interfere with each other by consuming any or all of these resources.

*File security*
: All users run code on the server with the same permissions (i.e., the same UID and GID). Any user who can write code for execution by the web server can read any files that are readable by the web server, no matter which user owns them. Similarly, any user who can write code for the web server can write any files that are writable by the web server, no matter which user owns them. Currently, it is not possible to run the `suEXEC` and `cgiwrap` extensions under mod_perl, and as mod_perl processes don't normally quit after servicing a request they cannot modify their UIDs and GIDs from request to request.

*Potential system compromise via user's code running on the web server*
: One of the possible solutions here is to use the *chroot(1)* or *jail(8)* mechanisms, which allow you to run subsystems isolated from the main system. So if a subsystem gets compromised, the whole system is still safe.

*Security of database connections*
: It's possible to hijack other users' DBI connections, and since all users can read each other's code, database usernames and passwords are visible to every user.

With all the problems described above, it's unwise to let users run their code under mod_perl on a shared server, unless they trust each other and follow strict guidelines to avoid interfering with each other's files and scripts (both of which are unlikely).

Note that there is no reason for an ISP not to run mod_perl applications that they control themselves. The dangers are only when they allow users to write their own mod_perl code. For example, an ISP might provide its users with value-added services such as guest books, hit counters, etc., that run under mod_perl. If the ISP provides code and data, which are not directly accessible by the users, they can still benefit from the performance gains offered by mod_perl.

mod_perl 2.0 improves the situation, since it allows a pool of Perl interpreters to be dedicated to a single virtual host. It is possible to set the UIDs and GIDs of these interpreters to be those of the user for which the virtual host is configured, so users can operate within their own protected spaces and are unable to interfere with other users.

## Users Sharing a Single Machine

A better approach is to give each user a dedicated web server, still running on the same machine.

Now each server can run under its owners' permissions, thus protecting users from each other. Unfortunately, this doesn't address the other considerations raised in the

previous setup approach. In Chapter 14 we discussed various techniques of limiting resource usage, but users will be able to override those limitations from within their code and you will have to trust your users not to do that.

Also, this scenario introduces a new problem. If an ISP uses named virtual hosts (all using the same IP address), what differs between users is the port to which their servers listen. The main frontend server will dispatch the requests to the various users' backend servers based on the port given to each user. If users are allowed to modify their parts of the server's *httpd.conf* file, it's possible that user A could adjust the server configuration to listen to the same port that user B's server is supposed to be listening to. User A's Apache server cannot bind to the same port while user B's server is running, but if the machine is rebooted at some point, it's possible that user A could take over the port allocated to user B. Now all the traffic that was supposed to go to user B will go to user A's server instead. User B's server will fail to start at all. Of course, ugly things like this will quickly be discovered, but not before some damage has been done.

If you have chosen this server-sharing technique, you must provide your clients with:

- Shutdown and startup scripts installed together with the rest of your daemon startup scripts (e.g., */etc/rc.d* directory), so that when you reboot your machine the users' servers will be properly shut down and restarted. Of course, you should make sure that the server will start under the UID of the user to whom it belongs.
- Rewrite rules in the frontend server. Since users cannot bind to port 80 in this scenario, they must bind to ports above 1024. The frontend server should rewrite each request to the correct backend server.
- Dedicated ports for each user. You must also ensure that users will use only the ports they are given. You can either trust your users or use special tools that ensure that. One such tool is called *cbs*; its documentation can be found at *http://www.epita.fr/~flav/cbs/doc/html*.

## Giving Each User a Separate Machine (Colocation)

A much better and simpler (but costly) solution is *colocation*. Let the users hook their (or your) standalone machines into your network, and forget about the issues raised in the previously suggested setups. Of course, either the users or you will have to undertake all the system administration. Many ISPs make sure only that the machine is up and running and delegate the rest of the system-administration chores to their users.

## Giving Each User a Virtual Machine

If users cannot afford dedicated machines, it's possible to provide each user with a virtual machine, assuming that you have a very powerful server that can run a few virtual machines on the same hardware.

There are a number of virtual-machine technologies, both commercial and open source. Here are some of them:

- The User-Mode Linux kernel gives you a virtual machine that may have different hardware and software virtual resources than the physical computer. Disk storage for the virtual machine is entirely contained inside a single file on the physical machine. You can assign your virtual machine only the hardware access you want it to have. With properly limited access, nothing you do on the virtual machine can change or damage your real computer or its software.

  If you want to completely protect one user from another and yourself from your users, this is yet another alternative to the solutions suggested at the beginning of this chapter.

  For more information, visit the home page of the project at *http://user-mode-linux.sourceforge.net/*.

- VMWare technology allows you to run a few instances of the same or different operating systems on the same machine. This technology comes in both open source and commercial flavors. The open source version is at *http://savannah.nongnu.org/projects/plex86/*. The commercial version is at *http://www.vmware.com/*.

  VMWare will allow you to run a separate OS for each of your clients on the same machine, assuming that you have enough hardware resources.

- freeVSD (*http://www.freevsd.org/*) is an open source project that enables ISPs to securely partition their physical servers into many virtual servers, each capable of running popular hosting applications such as Apache, *sendmail*, and MySQL.

- The S/390 IBM server is a great solution for huge ISPs, as it allows them to run hundreds of mod_perl servers while having only one box to maintain. The main drawback is its very high price. For more information, see *http://www.s390.ibm.com/linux/vif/*.