Designing Server Software for Zero Down-time

Jochen Topf
jochen@remote.org
http://www.remote.org/jochen/

2002-03-28

1 Introduction

Many papers have been written about fault-tolerant hardware design, about how to use clustering with hardware redundancies and automatic failover to achieve high reliability in the face of hardware failures. But, as any system administrator knows, most server breakdowns are not caused by hardware failures. More frequently the cause for system downtime is software, and quite often the reason is not a programming error, but bad design.

There are three distinct cases why a piece of server software might not be running even if it is bug-free:

(a) because the software didn't react well to a change in the environment, e.g. an unexpectedly high load

(b) because of a reconfiguration of the software by the system administrator, or

(c) because of software upgrades.

Reconfigurations and software upgrades are normally planned well before the downtime, but the world wide Internet has no real "slow" hours or days during which maintenance work would go unnoticed. Invariably customers will be affected.

The thoughts about these issues and about the solutions were born out of my work for a large ISP designing and developing an email system. Many of the ideas have been implemented in my POPular POP3 server [1], and I will use it throughout this paper as a practical example. The lessons learned can and should be applied to other types of server software as well.

2 Changes in the Environment

2.1 The Problem

Every system administrator knows this situation: The server is abysmally slow and when he examines the system, there are hundreds or even thousands of processes running. In many events the root of the problem cannot be determined because the machine is paging constantly and too slow to do real debugging. The mess of server software, database backend, CGIs or whatever is running on the machine is hopelessly tangled, giving no clue as to which part of the system is responsible, and which part is only a victim itself.

The typical strategy to get rid of the problem is to kill all server processes, to wait until the load on the machine has dropped to normal levels and to restart the service. Obviously, this is not the best strategy. It causes prolonged downtime of the service and, to make matters worse, it can trigger the Yo-Yo effect: When the server is restarted, all those people waiting for the web pages to show up and madly pushing the reload button on their browser will connect at the same time and will immediately push the server over the edge again starting the game anew.

The reasons for such abnormal conditions are manifold: It could be the Slashdot effect hitting a web server, the email mania around Christmas, a network failure that leads to longer session duration or a DOS (denial of service) attack by some blackhat. Or maybe it is just faulty client software.

You cannot defend against all those problems, because most of the time the problem is in the client or the network over which you do not have control. The only thing you can do is make your server bigger. But no matter how large your server is, there will always be a way of overloading this server as well. In the real world, of course, you never have enough money to make the server as big as you want it to be.

Dealing with the overload without manual intervention also allows the system administrator to use his time more efficiently and tackle the real underlying problem.

Looking at the situation more closely, you will see that the problem is not the overload itself — it is how it is handled. Most programs do not handle it at all, they just keep using more and more resources (memory, CPU cycles, disk bandwidth, ...) until one resource is exhausted. And then everything goes to hell.

2.2 The Solution

The solution is not really hard to find, and generally not really that difficult to implement either. The software has to detect the end of the resources nearing and stop using them.

Many operating systems allow the setting of resource limits on processes. Unfortunately, the problem is not one process, but the whole lot of them. The remedy used by the operating system is to kill the offending process, which is not really what we want.

For a typical forking network server, where each request is handled in a new process, a simple measure would be limiting the number of child processes. The web server Apache, for instance, does that. It is easily implemented, but not really that effective in practice. Typically there are more programs running on that machine than just Apache. A runaway CGI script can take the whole machine down.

2.3 Detecting Overload

Instead of (or in addition to) the process limit, some measure of the system load can be used to detect overload. On a typical Unix system the average system load of the last minute is easily available. Other potentially useful values include the I/O rates or the time needed to accomplish some typical task on the server.

One word of caution, though: Most system metrics are averaged over the last minute or so. If the system load rises sharply within a few seconds, it might already be too late to react when you find out that the system is slow. Only experience will help you find the right values for your system, where "the overload" begins and ends.

It is especially important to implement a mechanism like this if you have widely differing system loads over time. Otherwise you have to build your system so big that it can cope with the few seconds of overload every day, which can cost you a lot of money.

A typical system where this problem occurs is a mail server allowing access to mailboxes through the POP3 protocol. Many people configure their mail client to fetch mail every hour. In some popular clients this is implemented by accessing the mailbox every hour **on the hour**. For the server the result is a huge peak in access numbers every hour. Normally this might be handled well, but under exceptional circumstances — around Christmas time, or in addition to an existing network problem — it might be enough to regularly send the system over the brink.

2.4 What To Do When Overload Strikes?

Detecting the overload is only the first step; of course you will have to decide what to do if your system goes into overload. You can just refrain from accepting new connections and this is what will happen if Apache has reached its process limit. If the problem is short-lived, on the order of a few seconds maybe, your server will process the connection backlog when the overload goes away and everything is well. However, if the problem persists for a longer time, new connections will be lost at some point — and with a heavily loaded system, the "longer" time can actually be very short.

Lost connections mean error messages to the user, they result in service degradation, and they are actually making the problem worse, because many clients will immediately retry the connection or users will madly click the reload button. An obvious choice for a better solution would be to inform the user about the overload condition and ask him or her to come back in an hour. But many setups do not allow you to present meaningful messages to the user (and coming up with a meaningful message is difficult anyway if you do not know when the system will be usable again). Thus, the best solution depends very much on the kind of clients, servers and the protocols used.

2.5 Each Case Requires a Different Solution

On average over 80% of POP3 mailbox accesses are reading an empty mailbox. For users who check their mail very frequently (e.g. once every minute), this percentage is even higher. So for the special case of the POP3 server the reaction to an overloaded system could be to always show users an empty mailbox. This will greatly reduce the load on the system, because most of the load is generated by random disk accesses needed for opening the mailbox.

The presented solution is kind of a cheat, but if the overload condition is short-lived, people will just find new mails a few minutes later, the same result they would see if the mail server was slow in accepting a new mail from the sender. Remember that this is only a last-minute measure designed to keep the whole system from grinding to a screeching halt.

I have implemented this solution in the POPular POP3 server, and it has proved to work extremely well. It is, of course, a very specific solution, but it highlights the fact that some small and maybe unusual ideas can go a long way to keep the system running under unusual circumstances.

For a web server the solution might be something completely different. Maybe it can answer a request to a dynamic web page with the cached response, as it might be faster than accessing a database or going through a CGI. In many situations the page would be the same or only slightly different anyway.

Of course the system should not run in overload condition for long periods of time. It is only an emergency measure to keep the problem from getting out of hand, because, after all, you are not delivering the service you are supposed to deliver, even if nobody notices. It is the system administrators' job to check how often and why the system goes into overload and tune the system parameters or buy new hardware if problems persist.

2.6 Denial-of-Service Attacks

Defending against denial-of-service attacks is nearly impossible for the simple reason that, to the server, an attack looks like lots of normal legitimate request. Still, something can be done. A robust server can help the system administrator weather the attack, because instead of killing the machine, the service will keep running, albeit slower. Real customers will still be served. When the DOS attack is over, the system will return to normal conditions without the intervention of a human being.

3 Reconfigurations

With most programs today, to reconfigure a server means to edit a config file and then to restart the service or send the server a signal to reread the config file. Of course restarting the service means it is not available for a short period of time. When rereading the config file, the service interruption is generally minimal and if the network listening port is kept open and the time to reread the config file kept short enough, the client will not notice any interruption.

But there is a better model on how to reconfigure a running system: The Unix kernel. You don't have to reboot a machine to mount a file system, to add an IP number or change the routing table. Instead you send commands to the running kernel by invoking a system call, telling the kernel which part of its configuration to change.

There are a few user mode programs that do a similar thing. The USENET news server INN [2], for instance, can be controlled with the `ctlinnd` command. It can be used to create new newsgroups, change the running mode of the server, shut it down gracefully, and many other things.

Similarly, the BIND DNS server [3] can be reconfigured partially with the `ndc` command. Both these systems use a Unix domain socket to send commands and replies between the controlling program and the server, although BIND can be configured to use a TCP socket instead.

Both servers still use config files and some commands only instruct the server to reread the contents of a particular file. So it is kind of a hybrid system.

As systems get bigger and more complex it becomes ever more important to be able to change the configuration seamlessly. Large systems change all the time in response to customer demand or changes in the network. While the impact of one configuration change might be tiny, together they can lead to a lot of interruptions.

Is it not inconsistent that we frown upon operating systems which need a restart when an IP number changes, but at the same time live with server software that does the same?

Today more and more servers do not stand on their own but are part of a server farm. In such a setting it really makes much more sense to have a central server store the configuration and push all configuration changes out to the servers as they happen. It is much easier accomplished and less error-prone if we don't have to change configuration files and restart servers all the time.

3.1 The Root Problem

When writing a program that supports runtime configuration, a few things have to be considered. Static memory allocation, for instance, wont work any more, the configuration has to be broken down into small bits that are independent of each other, and some other issues. Most of them are easily solved, but there is one single problem that may keep many a programmer from using the scheme altogether:

Typical server software has to run as `root`, the only reason beeing that the process can bind a new socket to a low (< 1024) TCP port. Many programs will start out as a process owned by `root`, bind all the needed ports and then get rid of the privilege changing their user ID to some other user. Yet, because we want to be able to change everything on the fly while the program is running, even to be able to open new ports, we have to keep running as `root`. From a security point of view, this is unacceptable. Fortunately, there is a way around it.

3.2 File Descriptor Passing

File descriptor passing is a little known and little used feature of Unix domain sockets. After you have set up a connection between two programs using a Unix domain socket, those programs can exchange file descriptors among themselves. File descriptors are basically pointers to open files or sockets. After a file descriptor has been passed to the other process, two pointers exist to the same file or socket, one in each process. The effect is the same as when a process forks and execs a new program, but existing processes can share their file descriptors this way without one having to be created by the other.

Using this technique we can get rid of the "has to run as `root`" restriction, because we can have a little helper program running as `root` that will do the socket binding and send the bound socket to the server program. (We will later see that the little program can also help us with another problem.)

4 Software Upgrades

The third reason for service downtime is upgrading of the server software. Even if the software is designed to work well in overload conditions and is configurable while running, a software upgrade will in nearly every case be done in the old fashioned way of shutting down the old service, upgrading the software and starting the new service.

Frequently the effect is that the currently running sessions are lost, but even if it is done cleverly, it means that the listening sockets will close for a short time and if you have hundreds of connections per second to these ports, a few thousand customers might be unhappy.

Also, when doing upgrades in this way, many problems will lead to a longer downtime, because they have to be fixed while the server is down. Experienced system administrators know about the potential problems, like ports that can only be opened again after a timeout, and have their tricks to go around them, but the process is still very error-prone.

A better approach would be to install the new software, start it, configure it, switch over to the new version and only then stop the old program. Or maybe not stop the old program at all but keep it running in case there is a problem with the new version.

In the telecommunications world live upgrades have been the standard operating procedure for years. The software in phone switching equipment can be upgraded while the system is running. The old and new version of the software will happily run at the same time allowing a seamless upgrade.

For some reason, though, it is not done on a typical Unix system. To improve on the traditional way of upgrading software a few issues have to be considered.

The first problem is installing two versions of the same software on a system. Typical software package managers like RPM [4] do not really support this. Upgrading the software is generally done by uninstalling the old version and installing the new version. Of course you can always install from a tar file or use special packages that have versioning information in all path names.

4.1 Files

Even if you can install the binaries next to each other, the installation might conflict elsewhere. What about config files, log files, PID files, sockets, start and stop scripts and other files that a piece of software needs to function correctly? Let's have a look at them.

There are typically several kinds of files a server is using:

Config files We don't need config files, as long as we are doing run time configuration, but even if we do, we can split them up into different directories for each version or something similar.

Logs Logs are only a problem if you are not using syslog. If the software uses its own log file, both versions can write to it if they make sure to open the files in append mode.

PID file The new version will probably overwrite the PID file of the old version. Instead more than one PID file can be used. Although it seems counter-intuitive, it works well to include the process ID in the name of the PID file, e.g. use `/var/run/serv/serv.48176.pid` as the PID file for process ID 48176. Every time the software starts, it will create the appropriate PID file and then make a link from `/var/run/serv/serv.pid` to the real file. If a link already exists, it will be changed. If the software shuts down, it will remove its own PID file and also remove the link if the link points to a (now) non-existing file. So the canonical name `/var/run/serv/serv.pid` will exist only if at least one instance of the server software is running and it will point to the newest instance, which is very close to the normal semantics of the PID file.

Sockets and named pipes They have to be kept separate from each other. The same technique as with the PID files can be used.

Temp files We just have to make sure that the two versions are not using the same temporary files, i.e. by using file names with the process ID as part of the name or something similar. Fortunately, most software does that anyway.

Start/Stop scripts Generally only one script that starts the newest version is needed, because running two versions in parallel is only needed for the relatively short switch-over phase. Having two scripts for the old and new version is, of course, no problem if the version number is included in the script name.

Database files Almost every server software uses some kind of storage or database. For a mail server this is a number of mailboxes and the spool files, for a web server it consists of the web pages itself, for an LDAP server it might be a few DB-Files. Depending on the type of storage (file system vs. database, internal vs. external, read-only vs. read-write, availability of a locking mechanism, long-term vs. short-term etc.) either both versions of the program have to access the same storage or they can use different files, but it is highly dependent on the type of system and there is no "one size fits all" solution.

4.2 Listening Sockets

One large problem when running two versions of the same software together and moving the service from the old to the new version is the question of TCP ports. In the typical Unix server the program will open one or more sockets when it is starting up and bind them to the ports it wants to listen on. Rather sensibly, the Unix kernel prevents other programs from opening the same port so that nobody else can hijack connections destined for the first program.

But the new version of the same program that we are starting to replace the old version has to get to the port somehow. Closing the socket in the old program and only then opening it in the new program is not a solution, because for a short time the port would not be available leading to errors seen by the clients.

The classical way for a program to get an already open file or socket is to be executed by the program that has the file descriptor open. In our case the old version of the program would have to be instructed in some way to fork and execute the new version of the program.

There is another way: We have already seen in the last chapter that file descriptors can be passed through a Unix domain socket. This mechanism can be used to pass the sockets from the old to the new process, or a third process can be used as a mediator.

5 Putting it All Together

Making software more robust in the face of overload and allowing seamless reconfiguration and upgrades is only part of making a server really reliable. Equally important parts are hardware setup and fault-tolerant design of the software. Writing bug-free code is really hard, but software can at least be written in a way that small bugs do not have catastrophic consequences.

Another important issue is monitoring. Too many systems are black boxes, forcing the system administrator to use tools such as `ps` and system call tracers

to find out what is happening. To be able to react to problems in a running system any complex software needs some "peep-hole" specifically designed for the system to allow the system administrator to glimpse what is going on inside. Having extensive log files is equally important.

5.1 Implementation in the POPular Server

With the POPular POP3 server I have implemented many of these ideas in a consistent way, although not everything is perfect (yet). POPular is intended for large mail systems run in a distributed fashion on a server cluster. It is designed in a way that failures and overload conditions do not propagate from one host to another. The server uses the already described mechanisms of limiting the number of processes and reacting to high load on the machine.

The POPular server itself uses no config files. Nearly all aspects of the server configuration are changeable on the fly. New virtual servers can be added, new authentication modules loaded and unloaded, port numbers, timeouts, and many other variables can be changed.

Like INN and BIND the POPular POP3 server uses Unix domain sockets for sending commands to the running server. ASCII commands not unlike those in a shell are used, as they are easy for humans and programs to generate and easy to parse. Use of Unix domain sockets for configuring the server is easy to implement and easy to make secure, but it can only be a first step. A future version should support remote configuration.

POPular writes extensive log files, and their level of detail can be configured on the fly. Overload conditions and other problems are logged and clearly marked if system administrator intervention is necessary. Through the use of a shared memory region, a monitoring tool can be attached to the running server to look at statistics and watch what each session is doing.

Two or more versions of POPular can run at the same time. They share the same log file and mailbox storage but use different files for everything else. When upgrading, the old version can be instructed to keep running as long as there are sessions open and automatically quit when all work is done.

5.2 The Ringbearer

> *One Ring to rule them all, One Ring to find them,*
> *One Ring to bring them all and in the darkness bind them.*

For socket sharing between old and new version, POPular uses a tiny program called `ringd` which is started in the background as `root`. The program opens a Unix domain socket and waits for requests to arrive. If it gets a

request for a certain port, it will open a socket and bind the socket to the requested port. The port is then passed back to the requesting program. Security is achieved by file ownership of the Unix domain socket [1] and checking the configuration which ports are allowed to be opened.

This alone would suffice to solve the first problem: The server doesn't need to run as `root` any more. It also solves the problem of sharing bound sockets between different versions of the same software, because the `ringd` process keeps track of all requests and opened ports and will, when asked for the same port again, return the already opened port and not try to open a new one.

The `ringd` process keeps track of all the ports it opens and who requested them. A reference count makes sure that `ringd` closes unneeded ports.

References

[1] The POPular POP3 server is available under the GNU General Public License.
http://www.remote.org/jochen/mail/popular/

[2] INN: InterNetNews
http://www.isc.org/products/INN/

[3] ISC BIND
http://www.isc.org/products/BIND/

[4] Red Hat Package Manager
http://www.rpm.org/

[1] This doesn't work for some Unix versions like Solaris, but you can always put the socket in a directory with restrictive ownership and permissions