
1 GOOFED - Generic Object Oriented Flexible Extensible Daemon

Abstract

Perl ist eine tolle Sprache. Leider hat sie auch viele Abhängigkeiten. Besonders bei Embedding muss man im Allgemeinen eine ganze Perl-Installation "mitschleppen". Bei einem bestimmten Projekt musste ich ohne externe Dateien oder andere Abhängigkeiten auskommen. Um trotzdem Perl einsetzen zu können, musste alles in das Binary wandern. Die Probleme und Lösungen, die ich fand, sind im Folgenden beschrieben.

Wozu, weshalb und warum?

Das Projekt, für das Goofed entstand, bestand darin, den Radius-Server eines großen Internet-Providers in einigen Funktionen zu ersetzen. Da die Anforderungen durch viele Jahre hinweg sehr speziell wurden (ein typisches "Wir setzen den StandardPhänomen) und die alte Codebasis neuen Anforderungen nicht mehr gut gewachsen war, musste ein neuer Daemon her.

Recht früh kam die Idee, im Kern für die komplizierten Regeln eine High-Level-Skriptsprache zu verwenden und nur für das eigentliche Radius-Paket-Management C zu verwenden.

Perl schien zuerst an vielen Gründen zu scheitern: kompliziertes Build-System, viele Abhängigkeiten und Dateien, keine gute Kontrolle über den Speicherbedarf (Leaks wären tödlich für einen Daemon, der meherre Monate durchhalten muss) und Last-not-Least stand zu befürchten, daß die Server-Abteilung die Lösung sofort ablehnen würde, sobald bekannt würde, daß sie Perl benutzt (typisches "Real-World - macht keinen SinnPhänomen).

Eckpunkte der Implementierung

Trotz dieser Gegenanzeigen versprach Perl auch wichtige Vorteile, der wichtigste war das Vorhandensein einer grossen Codebasis und viele Module. Das und Perl selbst führen meist sehr schnell zu verwertbaren Ergebnissen.

Um die Anforderungen zu erfüllen, habe ich folgende Entscheidungen gefällt:

Einzelnes Binary

Bis auf die Konfigurationsdateien sollte alles - Perl-Interpreter, Module, Bibliotheksdateien - in einer einzigen Datei sein und keine externen Dateien benötigen oder anlegen.

Dies schließt leider das (ansonsten hervorragende) PAR-Modul aus, da dieses alle Bibliotheksdateien zwar in eine Datei packt, zur Laufzeit diese aber lediglich entpackt und entsprechend viele Dateien schreibt.

Festgelegte Perl-Version

Die Perl-Version sollte hartverdrahtet werden. Der Grund liegt in vielen, kleinen Änderungen auch innerhalb stabiler Release-Zyklen, die zu Build-Problemen oder Fehlern zur Laufzeit führen können. *Never change a running system* war hier ausschlaggebend.

Ausserdem kann man, sobald man die Kontrolle über die Perl-Quellen hat, bedenkenlos Anpassungen vornehmen (ich nenne das die "Lizenz zum RumsauenRegel).

Es sollte möglich sein, den Perl-Kern upzugraden, wobei aber klar sein sollte, dass dies Veränderungen am Quellcode nach sich zieht.

Festgelegte Module und festgelegte Versionen

Ähnliches gilt für Module: es sollte möglich sein, fast beliebige Module einzubauen, aber auch diese sollten in einer festen Version kommen.

Perl-Module auf CPAN haben - seien wir ehrlich - fast immer kleine und große Bugs. Meistens kann man mit ihnen leben oder Workarounds finden - häufig muss man dazu auf Interna zugreifen, die sich von Version zu Version ändern.

"Normales" Build-System, kein Configure

Configure nichtinteraktiv auszuführen ist kompliziert, und sehr viel kann schiefgehen. Ein einfaches Makefile fühlt sich wesentlich stabiler und einfacher wartbar an.

Da der Daemon sehr viele Anfragen erhält und in anderer Form weiterleitet, wäre es schön, wenn man die Bearbeitung der Anfragen verzahnen könnte. Dazu braucht man ein vernünftiges Event-System (der Name sagt schon - das Event-Modul) und etwas, mit dem man effizient und vor allem einfach Pseudoparallelität erreichen kann - Coroutinen, implementiert mit dem Coro-Modul.

Coro greift auf viele Perl-Interna zu und ist eigentlich recht experimentell, ich hatte aber Erfahrungen mit langlaufenden Servern und Coro und wusste daher, dass es sehr stabil läuft, sobald es läuft.

Daß das Umfeld stabil ist, gab den Ausschlag.

Implementierung

Nachdem ein paar grundlegende Gedanken gedacht waren, konnte ich an die Implementierung gehen. Flexibilität ist alles, wenn etwas schiefgeht.

Perl? Microperl!

Wenig bekannt ist, daß es auch eine Microperl-"Distribution" gibt, die ein sehr kleines und portables Perl verspricht - ohne Module, ohne Systemabhängigkeiten und ohne interaktives Configure.

Die "Distribution" ist sehr klein: sie besteht nur aus den drei Dateien `README.micro`, `Makefile.micro` und `uconfig.sh`. Daher ist sie bei Perl gleich mit dabei.

Ich habe das Makefile und die nötigen Perl-Sourcen in ein Unterverzeichnis kopiert und dann die Konfiguration, die standardmäßig nur ISO-C89 + einige wenige Erweiterungen wie `rename()` benötigt, solange getuned, bis ich Zugriff auf die meisten POSIX-Funktionen hatte.

Dazu habe ich die CFLAGS von:

```
-DPERL_CORE -DPERL_MICRO -DSTANDARD_C -DPERL_USE_SAFE_PUTENV
```

auf

```
-DPERL_CORE=1 -DPERL_MICRO=1
```

geändert und die Konfiguration in `uconfig.sh` getuned, compiliert und solange iteriert, bis ich zum Link-Stadium kam.

Danach hatte ich folgende Dateien in meinem `uperl`-Verzeichnis:

<code>EXTERN.h</code>	<code>globals.c</code>	<code>opnames.h</code>	<code>pp_hot.c</code>	<code>sv.h</code>
<code>INTERN.h</code>	<code>gv.c</code>	<code>pad.c</code>	<code>pp_pack.c</code>	<code>taint.c</code>
<code>Makefile</code>	<code>gv.h</code>	<code>pad.h</code>	<code>pp_proto.h</code>	<code>thrdvar.h</code>
<code>XSUB.h</code>	<code>handy.h</code>	<code>patchlevel.h</code>	<code>pp_sort.c</code>	<code>thread.h</code>
<code>av.c</code>	<code>hv.c</code>	<code>perl.c</code>	<code>pp_sys.c</code>	<code>toke.c</code>
<code>av.h</code>	<code>hv.h</code>	<code>perl.h</code>	<code>proto.h</code>	<code>uconfig.sh</code>
<code>config_h.SH</code>	<code>intrpvar.h</code>	<code>perlapi.c</code>	<code>reentr.c</code>	<code>universal.c</code>
<code>configpm</code>	<code>iperlsys.h</code>	<code>perlapi.h</code>	<code>reentr.h</code>	<code>unixish.h</code>
<code>cop.h</code>	<code>keywords.h</code>	<code>perlio.c</code>	<code>reentr.inc</code>	<code>utf8.c</code>
<code>cv.h</code>	<code>locale.c</code>	<code>perlio.h</code>	<code>regcomp.c</code>	<code>utf8.h</code>
<code>deb.c</code>	<code>mg.c</code>	<code>perliol.h</code>	<code>regcomp.h</code>	<code>util.c</code>
<code>doio.c</code>	<code>mg.h</code>	<code>perlsdio.h</code>	<code>regexec.c</code>	<code>util.h</code>
<code>doop.c</code>	<code>miniperlmain.c</code>	<code>perlvars.h</code>	<code>regexp.h</code>	<code>warnings.h</code>
<code>dump.c</code>	<code>nostdio.h</code>	<code>perly.c</code>	<code>regnodes.h</code>	<code>xsutils.c</code>
<code>embed.h</code>	<code>numeric.c</code>	<code>perly.h</code>	<code>run.c</code>	
<code>embedvar.h</code>	<code>op.c</code>	<code>pp.c</code>	<code>scope.c</code>	
<code>ext.libs</code>	<code>op.h</code>	<code>pp.h</code>	<code>scope.h</code>	
<code>form.h</code>	<code>opcode.h</code>	<code>pp_ctl.c</code>	<code>sv.c</code>	

Beim Bauen bekommt man die `libuperl.a`, das Microperl-Äquivalent zur `libperl` und `microperl`, das zum Bauen benötigt wird und auch beim weiteren Build-Prozess hilfreich sein kann. `microperl` wird z.B. benutzt, um aus der `uconfig.sh` das `Config.pm`-Modul zu basteln.

Mit `microperl`, `libuperl.a`, `Config.pm` und den Header-Dateien hat man alles, was man zum Embedden benötigt.

Dieser Schritt war relativ leicht. Er wäre noch leichter gewesen, wenn ich auf die meisten esoterischen Funktionen wie `getpwnam`, `mkdir` oder Signal-Handling verzichtet hätte, da ich fast alles über die `POSIX`- und `Event`-Module erreichen kann, und im Notfall auch auf C zurückgreifen könnte. Im absoluten Notfall.

Bibliotheksdateien

Im nächsten Schritt geht es um die Bibliotheksdateien, genaugenommen das *lib*-Verzeichnis, in dem sich die `.pm`-Dateien befinden. Theoretisch könnte man Microperl ganz ohne dieses betreiben: das Ergebnis wäre jedoch kaum noch "Perl" zu nennen, da auch Pragmas als Module implementiert sind. Und wie man an anderen, einfacher embeddbaren Sprachen sieht, ist Perl ohne Standardbibliothek eine nutzlosere embeddete Sprache, als man denkt.

Üblicherweise sind die `.pm`-Dateien in den jeweiligen Modul-Distributionen, aber bei Perl sind sie schon fertig abgepackt im `lib`-Verzeichnis.

Das größte Problem war es, alle Dateien und Unterverzeichnisse in CVS einzuchecken, nachdem ich sie aus der Perl-Distribution kopiert hatte (`cp -rp ~/../perl-5.8.x/lib lib-perl`).

Fragt sich noch, wie die Dateien in das Executable kommen. Dafür sorgt folgende Regel im `Makefile`:

```
UPERL = uperl/microperl -Ilib-perl

src/libfiles.c: s/gen_libfiles extensions uperl/microperl
    $(UPERL) s/gen_libfiles >$@
```

Die sorgt dafür, daß das Perl-Skript `s/gen_libfiles` aus allen Dateien die Datei `src/libfiles.c` generiert (`s` heißen bei mir Verzeichnisse mit Skripten, eine alte Amiga-Gewohnheit).

Die erzeugte Datei sieht so aus:

```
const char *embed_files[] = {
    /* 0 */
    "\025Attribute/Handlers.pm\000\000\030Apackage Attribute::Handlers;\01...
    " "data eq 'ARRAY';\012\011\011\011\\$data = [ \\$data ] unless \\$wa...
    ...
```

```

/* 1 */
"\007Carp.pm\000\000\016\017package Carp;\012\012our $VERSION = '1.01'...
...
...
,
};

const int embed_file_num = 208;

```

Die Datei ist 1139622 Bytes groß und mußte hier leider zweidimensional gekürzt werden. Wie man sofort sieht, folgt sie C99 und nicht C89, aber das habe ich in Kauf genommen.

Was man nicht sofort sieht: Alle Dateien werden in einem großen String-Feld gespeichert, pro Datei ein String. Am Anfang steht die Länge des Dateinamens, der Dateiname, danach die Länge der Datei und danach der Inhalt. Dieses Format habe ich gewählt, damit man die Datei schnell mittels binärer Suche finden kann und nicht einmal `strlen` aufrufen muß, bzw. auch binäre Dateien ablegen kann.

Die Funktion zum Suchen eines Eintrages sieht so aus (etwas umformatiert, damit sie besser paßt):

```

SV *
libfile(char *path)
  PREINIT:
  const char *s = 0, *f;
  int c, l = 0, m, r = embed_file_num;
  CODE:
  do
  {
    m = (l + r) >> 1;
    f = embed_files[m];
    c = strncmp (f + 1, path, *f);

    if (c > 0)      r = m - 1;
    else if (c < 0) l = m + 1;
    else
    {
      s = f + 1 + f[0];
      l = ((unsigned char)s[0] << 24) | ((unsigned char)s[1] << 16)
        | ((unsigned char)s[2] << 8) | ((unsigned char)s[3]      );
      s += 4;
      break;
    }
  }

```

```
    }
  }
  while (l <= r);

  RETVAL = s ? newSVpvn (s, l) : &PL_sv_undef;
  OUTPUT:
  RETVAL
```

Sie implementiert eine ganz normale binäre Suche und liefert den Inhalt der Datei als String, oder bei Nichtfinden `undef`. Ursprünglich habe ich eine lineare Suche benutzt (bei Rapid Prototyping bleibt manchmal etwas auf der Strecke, wenn man interessantere Stadien der Entwicklung erreichen will und eine binäre Suche auch nach Jahrzehnten Programmiererfahrung nicht auf Anhieb hinkriegt) und erwartet, das es keinen Unterschied macht. Die binäre Suche macht das Ergebnis dennoch einige Prozent schneller beim Starten, also scheint Perl beim Parsen recht flott zu sein.

Das Erzeugen der C-Strings aus den Dateien hat sich übrigens als weit komplizierter herausgestellt, als ich dachte. Zum einen muß man solche nervigen Sachen wie die maximale Zeilenlänge (4096) beachten, zum anderen ist das Quoting garnicht so einfach. Folgende Funktion quoted den String:

```
# print a c-escaped string
sub str {
  my $str = $_[0];
  while (length $str) {
    local $_ = substr $str, 0, 2000, ""; # 4096 is ISO-C max.
    s/\\/\\\\/g;
    s/([\x20-\x7e])/sprintf "\\%03o", ord $1/ge;
    s/"\\/"/g;
    s/\\x0a/\\n/g;
    print "  \"$_\"\\n";
  }
  print "  ,\\n";
}
```

Das andere Problem war, daß ich die Unmengen an Dokumentation nicht unbedingt im fertigen Programm brauche. Die Lösung ist `Pod::Stripper`, das sich der Benutzung allerdings widersetzt, da es unbedingt einen `FileHandle` möchte und `String-Streams` in `Microperl` nicht zur Verfügung stehen:

```
open my $ifh, "<", "$base/$path$_"
  or die "$base/$path$_: $!";
```

```

pipe my ($r, $w);
if (fork == 0) {
    close $r;
    Pod::Stripper->new->parse_from_filehandle ($ifh, $w);
    exit;
} else {
    close $w;
}

local $/;
$file{"$path$_"} = <$r>;

```

Ein `fork()` pro Datei und Parsen ist zwar nicht kostenlos, aber ich baue ja nicht unter Cygwin. Auf meinem Rechner braucht es 3,5s, was leider etwas lang ist, da ich schlecht eine Makefile-Regel bauen kann, die von allen Dateien in `lib-perl` abhängt und daher jedesmal das Skript aufrufe, wenn ich linke.

Module embedden

Für `goofed` brauche ich im Moment folgende Module:

```

Compress-LZF Coro-Event Coro-State Crypt-Rijndael Crypt-Twofish
Cwd Data Digest Event Fcntl File Filter IO List MIME POSIX PerlIO
Pod-Stripper Socket Spread Sys Time attrs daemon re

```

Header und Library sind zwar genug, um diese zu bauen, aber Perl-Module wollen ein lauffähiges Perl und Zugriff auf Kleinkram wie `ExtUtils::MakeMaker`, damit man sie übersetzen kann.

Zum Glück bietet `ExtUtils::MakeMaker` extra Unterstützung für diesen Fall. Das `Extension-Makefile.PL` führe ich so aus:

```

$TOP/upperl/microperl -I$TOP/lib-perl Makefile.PL INSTALLDIRS=perl \
    PERL_CORE=1 PERL_SRC=$TOP/upperl PERL_LIB=$TOP/lib-perl

```

Anscheinend sind `PERL_CORE=1` und `SRC=` der Trick. Perl machts beim Übersetzen genauso und es funktioniert, mehr Gedanken habe ich mir nicht gemacht.

Zum Übersetzen verwende ich folgenden `make`-Aufruf:

```

make CC=$(CC) OPTIMIZE=$(OPTIMIZE) LINKTYPE=static CCCDLFLAGS= static

```

Das und noch mehr ist in einem Skript namens `s/make_ext` verewigt, das nach dem Perl-Vorbild `ext/util/make_ext` geschrieben wurde. Es macht wenig mehr als die beiden

Kommandos auszuführen: es baut automatisch das **Makefile**, falls es keines gibt und akzeptiert z.B. **clean** als Argument. Nichts, was man mit einem gescheiterten **Makefile** nicht sauberer lösen könnte.

Dies baut für jede Extension, die XS benutzt, eine **libextension.a**, gegen die man linken muss.

PM-Dateien von Erweiterungen Erweiterungen besitzen fast immer auch eigene **.pm**-Dateien, die in das fertige Executable gehören. Mein (nicht ganz perfekter) Weg, dies zu erreichen war, **make install** in den fertig gebauten Erweiterungsmodulen zu machen. Dies installiert die **.pm**-Dateien (und eventuell andere Bibliotheksdateien) im **lib-perl**-Verzeichnis, wo sie beim Bauen automatisch gefunden werden. Ich darf dann bloss nicht vergessen, sie ins CVS einzuchecken.

Linken und das Hauptprogramm

Das Linken geht relativ einfach:

```
goofed: src/main.o src/xsinit.o src/libfiles.o extensions
$(CC) $(CFLAGS) -o $@ src/xsinit.o \
    'find lib-perl/auto -name '*.a' -print' \
    src/main.o src/libfiles.o uperl/libuperl.a $(LIBS)
```

Es werden einfach alle statischen Libraries, **libuperl.a** und die übersetzten Dateien **libfiles.c**, **xsinit.c** und **main.c** zusammengelinkt.

main.c enthält das Hauptprogramm. Es macht im wesentlichen das gleiche, das auch Perl macht (hier ohne error-checking, das Dokument **perlembded** enthält genauere Erläuterungen zum Embedding allgemein). Darin sind zwei wichtige Aufrufe:

```
perl_parse (my_perl, xs_init, EMBED_ARGC, EMBED_ARGV, (char **) NULL);
perl_run (my_perl);
```

perl_parse parsed die Kommandozeilenargumente, genau wie **perl**, und initialisiert die eingelinkten Erweiterungsmodule.

Üblicherweise (d.h. heutzutage) werden Erweiterungen als Shared-Objects gebaut und dynamisch (mit **DynaLoader** oder **XSLoader**) geladen. Der **DynaLoader** generiert aus dem Package-Namen einen Dateinamen und versucht, diese Datei als Shared-Object zu öffnen. Danach sucht es darin eine Funktion namens **boot_Name**, z.B. **boot_Data__Dumper**, registriert sie bei Perl und führt sie aus.

Diese Funktion registriert alle XS-Funktionen als Perl-Funktionen und führt den **BOOT**:-Abschnitt aus.

Eine statisch eingelinkte Erweiterung kann nicht dynamisch nach einem Namen durchsucht werden, daher muss man die Funktionen "per Hand" bei Perl registrieren (das Aufrufen geschieht erst beim Laden des Perl-Moduls). Dies erledigt die `xs_init`-Funktion in der Datei `xsinit.c`.

Diese Datei kann man sich mit Hilfe des `ExtUtils::Embed`-Moduls schreiben lassen (siehe `perlembed`), ich hatte aber wenig Glück, dies automatisch zu tun (manche Module waren doppelt, andere haben gefehlt). Seither Pflege ich sie per Hand. Die Datei sieht in etwa so aus:

```
#include "EXTERN.h"
#define PERL_IN_MINIPERLMAIN_C
#include "perl.h"

EXTERN_C void boot_Compress__LZF      (pTHX_ CV *cv);
...
EXTERN_C void boot_Spread             (pTHX_ CV *cv);
EXTERN_C void boot_daemon             (pTHX_ CV *cv);

EXTERN_C void
xs_init (pTHX)
{
    char *file = __FILE__;

    newXS ("daemon::bootstrap", boot_daemon, file);
    newXS ("Spread::bootstrap", boot_Spread, file);
    ...
    newXS ("Compress::LZF::bootstrap", boot_Compress__LZF, file);
}
```

Wenn das Parsing vorbei ist, baue ich noch `@ARGV` auf - mein Aufruf von `perl_parse` benutzt ja nicht die echten Kommandozeilenargumente - und rufe `perl_run` auf, das die Kontrolle endgültig an Perl abgibt.

Bootstrapping von Perl

Welches sind nun die Argumente für den Perl-Interpreter?

```
/* this array contains the initial startup code for the perl interpreter */
char *embed_argv[EMBED_ARGC] = {
    "goofed",
    "-e",
    "\n"
```

```
" daemon->bootstrap (0);\n"
... weiterer Perl-Code ...
};
```

Das erste Argument ist der Name des Programms. Danach kommt eine `-e`-Option mit dem Startprogramm. Es sieht so aus:

```
daemon->bootstrap (0);
PerlIO::scalar->bootstrap (0);
@INC = sub {
    open my $fh, '<', \(daemon::libfile ($_[1]))
        or die "FATAL: loader can't in-memory stream\n";
    $fh;
};
require daemon;
daemon->init;
```

Die ersten beiden Zeilen initialisieren die beiden Module `daemon` (ein Modul, in das ich die meisten Spezial-C-Funktionen für `goofed` gepackt habe) und (ausgerechnet!) `PerlIO::scalar`.

Ersteres enthält die `libfile`-Funktion, mit dem ich die Perl-Sourcen bekomme und letzteres wird von Perl benötigt um Memory-Streams (`open FH, "<", \ $scalar`) zu supporten. Die beiden `bootstrap`-Funktionen registrieren allerdings *nur* die XS-Funktionen, die zugehörigen Perl-Module kann man zu diesem Zeitpunkt nicht laden.

Um das zu ermöglichen, überschreibe ich `@INC` mit einer Funktion. Versucht Perl nun eine Datei zu laden durchsucht es `@INC`, findet die Funktion und führt sie aus. Diese holt sich die entsprechende Datei und gibt einen Filehandle zurück. Perl liest und parsed sie dann.

Danach kann man die Perl-Module ganz normal mit `use` oder `require` laden, was auch prompt gemacht wird, indem das `daemon`-Modul geladen und dessen `init`-Funktion ausgeführt wird.

An diesem Punkt hat man ein Perl, in das man relativ einfach CPAN-Module einbinden kann und keinerlei Abhängigkeiten im Filesystem besitzt, schnell bootstrapped, Support für binäre Objekte (Bilder etc.) hat und die gesamte Power von Perl zur Verfügung stellt.

Benutzung

Nachdem ein solcher Aufwand getrieben wurde, wollte ich noch die Frage klären, ob es sich lohnt, Perl statt C zu benutzen.

Die Hauptaufgabe war es, einen Radius-Server zu bauen (wobei man mit `goofed` natürlich noch andere Aufgaben erfüllen kann, es dient also eher als Anwendungsplattform :)

Das Parsen von Radius-Paketen ist eine Byte-Pfrieemelei, braucht aber sehr viel Konfigurationsdaten. Das komplette Modul zum Parsen und Erzeugen von Radius-Paketen ist (inklusive reichlich Dokumentation) 307 Zeilen lang. Ähnlicher Code im *Merit-Radiusd* besteht aus ca. 8000 Zeilen C. Ohne Dokumentation.

Allein hier hat sich Perl gelohnt. Richtig interessant wird es allerdings bei den speziellen Anforderungen: eine Authentifizierung benötigt ca. zehn LDAP-Anfragen pro Request. Im alten Radius-Daemon waren diese blockierend und haben den Großteil der Realzeit verbraucht.

In `goofed` sehen sie zuerst auch blockierend aus:

```
my $ldap_user = $self->{ldap}->search ("uuRadiusUser=$user, uuRadiusRealm=$realm, $ldap_root
    or return $self->nak ("user unknown");
... tue etwas
my $ldap_profile = $self->{ldap}->search ("uuRadiusProfile=$profile, $ldap_root");
... tue noch mehr
$self->{ldap}->search ("uuRadiusNAS=$nas_ip, uuRadiusRealm=$realm, $ldap_root")
    or return $self->nak ("roaming not allowed");
... usw.
```

Auch hier wird eine Anfrage gebastelt und auf das Ergebnis gewartet. Der Unterschied ist, das die Anfrage asynchron verschickt wird und in der gleichen Zeit andere Radius-Pakete entgegengenommen werden können und Anfragen gestellt bzw. verarbeitet werden können.

Da das LDAP-Protokoll selbst asynchron ist, besteht bei einem entsprechend intelligenten LDAP-Server (ich kenne keinen) sogar die Möglichkeit, einfache Anfragen sofort zu beantworten während komplizierte im Hintergrund erledigt werden.

Das LDAP-Modul verwendet nicht das `Net:LDAP`-Modul von CPAN, da ich dieses nicht stabil mit `OpenLDAP` zusammenbringen konnte, bzw. es sich extrem gegen Event-basierte Programmierung gewehrt hat. Es benutzt stattdessen direkt die `libldap` von `OpenLDAP` (durch `Xs`-Funktionen in `daemon.xs` implementiert).

Das erste, was es macht, ist, eine Verbindung zum LDAP-Server aufzumachen und einen Event-Handler darauf zu binden:

```
$self->{ldap} = new ldap::conn $self->{host}, $self->{port} || 389
    or die "unable to connect to ldap server $self->{host}:$self->{port}: $!\n";
...
$self->{w} = Event->io (
    fd => $self->{ldap}->fd,
    poll => 'r',
    cb => [$self, "_recvresp"],
);
```

Normalerweise ist es mein Stil, Callback-Funktionen als anonyme Sub-Funktionen auszuliegen. Nur leider gibt dies mit aktuellen Coro-Versionen manchmal Memory-Leaks, die ich nichtmal mit `valgrind` finde.

Die `_recvresp`-Methode holt einen Ergebnis-Record vom Server ab, baut bei Abbruch die Verbindung neu auf, und ruft den entsprechenden Callback für die ursprüngliche Anfrage auf.

Anfragen sind etwas aufwendiger: Einerseits mag es der LDAP-Server nicht, wenn zu viele Anfragen gleichzeitig ausstehen (warum, weiss ich nicht, aber da wenige Anwendungen asynchron Anfragen stellen, ist dies vielleicht nicht gut getestet): es treten Fehler auf und sie müssen wiederholt werden; andererseits darf immer nur einer gleichzeitig eine Anfrage stellen (die C-Library ist zwar *reentrant*, nicht aber der FileHandle, auf den sie schreibt).

Sowohl für die globale Limitierung als auch für das gleichzeitige schreiben verwende ich eine `Coro::Semaphore`:

```
# in sub new
my $self = bless {
    host => $host,
    lock => (new Coro::Semaphore),
    limit => (new Coro::Semaphore 20),
    %arg,
}, $class;
```

Der Abfragecode macht dann:

```
$self->{limit}->down; # limit # of simult. requests
my $sem = new Coro::Semaphore 0;
$self->{lock}->down; # ensure that there is only a single writer
```

Uff, drei Semaphore! Die erste sorgt dafür, dass nur eine bestimmte Anzahl von Abfragen gleichzeitig ausstehen können. Sie wird erst nach dem Lesen des Ergebnisses wieder freigegeben.

Die zweite Semaphore wird schon gelockt erzeugt und wird nur benutzt, um auf das Ergebnis zu warten (mehr dazu gleich).

Und die dritte Semaphore-Operation lässt immer nur einen gleichzeitig Anfragen zum Server schicken.

Dies geschieht danach: (Achtung: stark vereinfacht :)

```
my $id = $self->{ldap}->search ($base);
# registriere Callback
$self->{id}{$id} = sub {
    ... parse Ergebnis
```

```
    $sem->up;
}
```

Danach werden die Semaphoren wieder gelöst:

```
$self->{lock}->up;
$sem->down; # sleep till result is there
$self->{limit}->up;
```

Dies geschieht in umgekehrter Reihenfolge (das ist notwendig, um einen Deadlock zu verhindern) und definiert den Zeitablauf:

Zuerst wird die Schreibzugriff-Semaphore freigegeben, da die Anfrage mit `search` zum Server geschickt wurde und nun andere Anfragen schicken dürfen.

Danach wird auf das Ergebnis gewartet: Die `$sem`-Semaphore ist gelockt und wird erst im Callback wieder gelöst (`$sem->up`), wenn das Ergebnis eingetrudelt ist.

Zuletzt wird die Semaphore, die die maximale Anzahl ausstehender Anfragen begrenzt, gelöst, da ja die Anfrage durchgeführt wurde und das Ergebnis vorliegt.

Das sind 177 Zeilen (die XS-Schnittstelle zur OpenLDAP-Library nicht mitgezählt). Dadurch ist dir LDAP-Funktionalität abgekapselt und leicht einzeln testbar.

Der Vorteil ist, daß man den eigentlichen Authentifizierungscode sehr natürlich schreiben kann, nämlich sequentiell ohne zwischendurch irgendwelche Locks beachten zu müssen oder irgendwelche Zustände zwischenspeichern, und trotzdem in den Genuss voller Parallelität von LDAP-Server und Radiusd gelangt. In der Praxis ist deshalb der LDAP-Server der Engpass bei der Authentifizierung.

Hinweise und Verweise

Quellcode für `goofed` ist leider nicht frei erhältlich. Ich hoffe aber, das man ähnliches jetzt selbst nachbauen kann.

Handarbeit ist nicht immer das Beste. Man kann auch einfacher in den Genuß von einzelnen Binaries kommen. Z.B. mit dem `PAR`-Modul (von CPAN), das aus allen Modulen eines Programms eine Art Archiv mit Perl-Lader bauen kann. Es kommt dem obigen sehr nahe, da es ebenfalls ein einzelnes Binary erzeugen kann. Allerdings entpackt es zur Laufzeit die Dateien und kann auch keine Bibliotheken einlinken (Shared Objects bleiben Shared Objects).

Mit `makeaperl` aus der Perl-Distribution kann man sich ein statisches Perl-Binary mit allen Modulen bauen, die man braucht. Dies beinhaltet aber nicht die Perl-Quellen dazu.