
1 Freenet - Informationsfreiheit ohne Zensur, aber mit Perl

Abstract

Das Freenet-Projekt (<http://www.freenetproject.org>) hat es sich zur Aufgabe gemacht, jedem die Möglichkeit freier Meinungsäußerung zu garantieren. Neben der Implementierung einer Lösung für dieses Problem durch das Projekt geht es in diesem Artikel darum, wie man in Perl selbst Informationen herunterlädt bzw. veröffentlicht.

Einführung

Das Freenet-Projekt (<http://www.freenetproject.org>) hat es sich zur Aufgabe gemacht, jedem die Möglichkeit freier Meinungsäußerung zu garantieren. Als Mittel zur Umsetzung hat es ein virtuelles Netzwerk gewählt, in dem es nicht realistisch möglich sein wird, Einspeisung von neuen Daten zu verhindern oder den Urheber von Veröffentlichungen ausfindig zu machen, solange er dies nicht selbst zugibt. Darüberhinaus ist es für Betreiber von Freenet-Knoten nicht möglich, zu wissen, was auf ihrer Festplatte gespeichert wurde oder wo die Daten herkommen, so daß es ab einer bestimmten Netzwerkgröße nicht mehr möglich ist, dem Betreiber einer Freenet-Node nachzuweisen, daß er illegale Inhalte besitzt bzw. weitergegeben hat oder davon gewusst hat.

In China beispielsweise wird eine frühe Version von Freenet für ein rein chinesisches Freenet-Netzwerk aktiv benutzt, um politische Inhalte verbreiten zu können, ohne Maßnahmen von der Regierung zu befürchten.

Freenet für den Benutzer

Freenet speichert nur relativ kleine Dokumente ($\leq 1\text{MB}$) am Stück. Diese Dokumente können beliebige Daten enthalten, z.B. HTML-Seiten. Diese Datenblöcke können mit einem für jeden Block eindeutigen Schlüssel abgerufen werden, z.B. in einem Web-Browser.

Hier ist ein Beispiel für eine solche HTML-Seite:

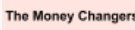













<- http://129.13.162.73:8888/SSK@Sc6qV~D6iFhaYord6HtbjJ8MaEYPaGm/YoYo//Controversy.html

YoYo!

YoYo! Freenet Directory

IIP Registered Nick: Reskill | E-Mail: reskill@iipmail.net

Controversy

<p>The Money Changers </p> <p>review: All about the US Federal Reserve, with history and lots of quotes etc - appears to have been taken, in part at least, from the WWW. description: Explains the history of the reserve banking system and why everyday people (including Bill Gates) continue to suffer at the hands of a very few.</p> <p> Edition Last Update: 20031230 Update Avg: 0.43</p>	<p>Pussy Galore </p> <p>review: Cats, Princess Diana, and an interesting mutating picture of G.W. Bush. description: the site that overflows with the milk of human kindness. His Grace Duke Morbid (also Knight of the Order of the Garter and Tampon) shares his reflections on life. For the less educated visitor there are plenty of pictures. PG is dedicated to Dead Diana and all who have known her in the Biblical sense.</p> <p> DBR Last Update: 20040507 Update Avg: 0.32</p>
<p>Thought Crime </p> <p>review: Making the most of freedom of speech description: Preserving unpopular content and controversial opinions</p> <p> Edition Last Update: 20040310 Update Avg: 0.04</p>	<p>Daybreak </p> <p>review: Many Nationalist links and resources. description: White Nationalist News and Views.</p> <p> Edition Last Update: 20030702 Update Avg: 0.01</p>
<p>Requiem </p> <p>review: Lists some sites that can be retrieved in "Production Freenet", and probaby here also. description: No Description Available</p> <p> Edition Last Update: 20031007 Update Avg: 0</p>	<p>Watched </p> <p>review: A very nicely put together site offering suggestions on how to maintain your own privacy. description: Act collectively to prevent surveillance of the population. Use PPDs to keep Big Brother deaf and blind.</p> <p> Edition Last Update: 20031003 Update Avg: 0</p>
<p>The Daily Slander </p> <p>review: Sometimes horribly offensive, sometimes incredibly funny, this is freenet's premiere site for satire. description: You stink, in fact your whole family stinks, your wife is a slut and your kids are just a hair away from being Autistic. -</p>	<p>Subversive Bookwares </p> <p>review: A large collection of so called questionable materials and links to sites about privacy and piracy. description: Edition freesite only available on Freenet. It contains about 50 books in assorted electronic formats and they are be</p>

Request sent

Die hier verwendete URL ist:

http://129.13.162.73:8888/...
...SSK@Sc6qV~D6iFhaYord6HtbjJ8MaEYPaGm/YoYo//Controversy.html

`http://129.13.162.73:8888` ist die Adresse meines Freenet-Daemons, üblicherweise `http://127.0.0.1:8888`, der Sicherheit wegen sollte man immer einen eigenen Freenet-Daemon laufen lassen. Der Schlüssel der "Freesite" ist `SSK@Sc6qV~D6iFhaYord6HtbjJ8MaEYPagM` und in dieser wird wiederum das Unterdokument `YoYo//Controversy.html` angezeigt.

Diese Seite ist Teil eines sehr bekannten Einsprungpunktes ins Freenet. Generell bevorzugt das Freenet-Projekt keine bestimmten Einsprungpunkte. Die Regel ist: kennt man den Einsprungpunkt nicht, findet man die Inhalte nicht; es ist sogar unmöglich zu bestimmen, wieviel oder welche Informationen im Freenet gespeichert sind: Ein Index a'la google ist prinzipbedingt nicht möglich.

Obwohl es einige Freenet-Spider und Directory-Freesites gibt ist der überwältigende Teil der Information im Freenet nicht darüber zu erreichen.

Neben HTML-Inhalten gibt es auch Mail-, Foren- und Chat-Systeme im Freenet.

Die Einspeisung von Daten kann von überall im Netz geschehen. Mehrfacheinspeisung ist möglich (und meistens notwendig) und führt nicht zur Duplizierung von Daten.

Natürlich gibts es auch einige Nachteile, die teilweise mit dem Design und teilweise mit der Implementierung zusammenhängen. So ist der einzige verfügbare Freenet-Daemon in Java implementiert und daher leider extrem unportabel (läuft praktisch nur auf Linux/x86 und Windows, da das benötigte Sun-Java-JDK auf anderen Plattformen nicht verfügbar oder veraltet ist), sehr langsam und vor allem speicherhungrig, und häufig auch sehr instabil.

Die aktuellen Versionen des Freenet-Daemons sind alle als "nicht sicher" deklariert, da erst die Release 1.0 verspricht, alle Ideen umgesetzt zu haben, wovon das Projekt noch 0.5 Versionen entfernt ist :)

Designbedingt ist das Freenet eher langsam (extrem hohe Latenz, akzeptabler Durchsatz) und für interaktive Benutzung teilweise ungeeignet, bzw. eine echte Geduldprobe.

Durch die Art der Speicherung kann nicht garantiert werden, daß Informationen überhaupt wiedergefunden werden. "Unpopuläre" Information verschwindet, sofern sie nicht regelmäßig eingespeist oder abgerufen wird, automatisch wieder aus dem Freenet.

Die Freenet-Architektur

Herkömmliche File-Sharing-Netzwerke haben zwei große Nachteile: Entweder sind sie nicht wirklich anonym (selbst anonymisierende Netzwerke oder Proxies sind üblicherweise nicht vor Zugriff durch Regierungen sicher (ein bekanntes Beispiel ist *JAP*, `http://anon.inf.tu-dresden.de`, das zwar vollmundig vollkommene Anonymität garantiert, aber schon einmal durch die Polizei gezwungen wurde, Log-Informationen herauszugeben, wozu die Betreiber nach deutschem Recht verpflichtet sind). Oder sie skalieren nicht, da sie Broadcast-Algorithmen benutzen, die größere Netzwerke von vorneherein ausschließen (bekanntes Beispiel dafür ist *gnutella*).

Das erste Problem umgeht Freenet (Achtung, vereinfacht!), indem es jeden Datenblock

hasht und aus diesem Hash einen Schlüssel für die Verschlüsselung generiert und die Daten damit verschlüsselt (sic).

Der Schlüssel wird ein zweitesmal gehasht. Dieser Hash wird zur eindeutigen ID des Datenblocks. Diese ID und der verschlüsselte Datenblock werden ins Freenet eingespeist. Dabei wird immer derselbe Key generiert (da er nur von den Daten abhängt), und der verschlüsselte Block ist ebenfalls immer gleich. Daher kann er problemlos mehrfach und von unterschiedlichen Parteien eingespeist werden ohne die Daten faktisch mehrfach im Freenet abzulegen.

Das ist sicher, da ein kryptographisch sicherer Hash (der die Grundlage des Systems bilden muß) nur in eine Richtung funktioniert: Aus der ID (= Hash des Hashes der Daten) kann nicht auf den Schlüssel geschlossen werden. Speichert also ein Netzknoten die Daten und die ID dazu, kann man die Daten zwar abrufen, jedoch nicht entschlüsseln. Selbst der Besitzer eines Knotens kann mit vollkommenen Wissen über alle Vorgänge seines Knotens die Inhalte nicht lesen.

Eine weitere Konsequenz dieses Verfahrens ist die Tatsache, das Dokumente niemals verändert werden können: eine Änderung bewirkt eine Änderung des Schlüssels und damit eine neue ID, praktisch eine völlig neue URL.

Möchte man z.B. das grüne Blatt der Freesite "Thought Crime" herunterladen so wird man mit folgendem Freenet-Key konfrontiert:

```
CHK@pfAv9IejYPLQwaTLXDguEkUhiNUMAwI ,blzDbhN~8Q28esq4JrfrWw
```

CHK steht für *Content-Hash-Key*, der häufigste Schlüsseltyp im Freenet, der das oben beschriebene Verfahren benutzt. Der CHK besteht aus zwei Teilen: der ID, unter der der Schlüssel im Freenet abgelegt wird (`pfAv9IejYPLQwaTLXDguEkUhiNUMAwI`) und dem Schlüssel, mit dem die Daten verschlüsselt wurden (`blzDbhN~8Q28esq4JrfrWw`), durch ein Komma getrennt.

Das sind übrigens (mehr oder weniger) base64-encodete Daten, dekodiert sieht der Schlüssel so aus:

```
CHK@
ID = a5f02ff487a360f2d0c1a4cb5c382e12452188d50c0302
Key = 6e5cc36e137ef10dbc7acab826b7d15b
```

(Man kann daraus tatsächlich ablesen, das der Datenblock 4k groß (0x0c am Ende der ID bedeutet 2**12) ist und Twofish benutzt. Aber derartige Details überläßt man lieber einem Perl-Modul).

Nur der erste Teil wird als Anfrage ins Freenet geschickt, der zweite Teil bleibt im lokalen Freenet-Daemon. Wird nun der Datenblock geliefert, kann er mit dem Schlüssel dekodiert werden und wird an den Benutzer geliefert.

Das zweite Problem (der Skalierbarkeit) wird durch lineare statt exponentielle Suche gelöst: mit jeder Anfrage wird eine *Hops-To-Live*-Angabe (HTL, ähnlich wie das TTL

in IPv4) verknüpft. Diese Zahl, die üblicherweise zwischen 5 und 25 (maximal) liegt, gibt an, wie viele Knoten die Anfrage maximal weitergeleitet wird. Jeder Knoten, der die Daten nicht lokal vorrätig hat, leitet sie an denjenigen Knoten weiter, der die Daten am wahrscheinlichsten hat. Da das Verfahren linear ist und nicht exponentiell, skaliert es ebenfalls linear mit der Netzwerkgröße.

Lädt man das Dokument, wird man mit Metadaten und "normalen" Daten konfrontiert. Metadaten:

```
Revision=1
EndPart
Document
Info.Format=image/png
End
```

Daten:

```
PNG^Z...
```

Nun stellt sich die Frage, wie man sicherstellt, daß Daten im Freenet bleiben, da es nur endliche Speicherkapazität hat. Die Antwort ist überraschend: Es geht nicht. Wenn ein Knoten sich entscheidet, Daten zu löschen oder Platz für neue Daten zu schaffen, gehen zwangsläufig andere Daten verloren. Dagegen hilft nur wiederholtes Einfügen und der Wunsch, daß die eigenen Daten beliebt genug sind, damit sie abgerufen werden und damit länger überleben.

Absolute Anonymität und nichtnachvollziehbarkeit von Transaktionen ist eben nicht vereinbar mit garantierter Datenspeicherung. Könnte man garantiert nachweisen, das ein Dokument existiert, so wäre es für einen Knotenbetreiber schlecht möglich, Wissen über die Art der Daten abzustreiten, die er speichert.

SSK-Schlüssel Der zweite gebräuchliche Schlüsseltyp, mit dem man im Freenet konfrontiert wird, ist ein SSK-Schlüssel (SSK steht für *Signed Subspace Key*). Hier ist einer:

```
SSK@00hVDWuti bbBMbXmbxNXWOM6YFoPAgM
```

Zwei Dinge unterscheiden Sie von CHK-Schlüsseln:

Erstens ist ein SSK-Schlüssel eigentlich ein Schlüsselpaar, ein Schlüssel (der private Schlüssel) muss für das Einfügen von Daten benutzt werden, der zweite Schlüssel (der öffentliche Key) wird für das Auslesen der Daten benutzt. Da die Daten signiert sind, kann nur der oder die Besitzer des privaten Schlüssels Daten unter diesem einfügen, während die Besitzer des öffentlichen Schlüssels überprüfen können, ob die Daten tatsächlich mit dem richtigen Schlüssel signiert wurden.

Zweitens ist die ID, unter dem diese Daten abgelegt werden, nicht von den Daten sondern nur vom "Namen" (dem Key) abhängig, man kann also URIs erzeugen auf Dateien, die noch nicht eingefügt wurden.

Diese Art Schlüssel ist relativ ineffizient und kann keine großen Datenmengen speichern (<= 32KB!), SSKs werden also vorwiegend für Weiterleitungen auf CHKs benutzt.

Der obige SSK gehört übrigens zur "Freenet Explained"-Freesite, die die einzelnen Schlüsseltypen (und mehr) erklärt. Finden kann man sie hier:

```
SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//
```

bzw., wenn man einen Freenet-Daemon am Laufen hat, hier:

```
http://127.0.0.1:8888/SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//
```

Und das ganze in Perl

Das ganze wäre relativ witzlos, wenn man nur irgendwelche langweiligen Tools und/oder Java benutzen kann. Es gibt zwei Perl-Module (bzw. Modulfamilien), `Net::Freenet::FCP` (<http://www.sf.net/projects/perlfcf>) und `Net::FCP`.

Ersteres ist älter (und vielleicht besser benannt) und konzentriert sich mehr auf die Verarbeitung von Metadaten, ist aber recht schwach beim eigentlichen Protokoll-Handling. Letzteres ist sehr gut beim Protokoll-Handling und stark beim Kodieren und Dekodieren von Daten, überläßt das Metadatenhandling aber dem Programmierer.

`Net::FCP` stammt von mir und ist entstanden, weil ich das andere Modul, das es nur im Freenet gab, nicht herunterladen konnte (tja, so ist das eben als Freenetter). Daher gehe ich auch nur auf dieses Modul ein :)

Installation

Zuerst braucht man einen Freenet-Daemon (<http://www.freenetproject.org>). Die Installation ist unglaublich komplex (unter Windows gibt es einen Installer, unter Unix ein paar Skripte). Der Daemon braucht eine Weile, bis er warm wird (Minuten bis Stunden), was man durch Surf-Versuche unterstützen sollte.

Das `Net::FCP`-Paket gibt es ganz normal per CPAN.

Einfache Abfragen

Die Abkürzung *FCP* steht für *Freenet Client Protocol* und ist das Protokoll zwischen Freenet-Anwendungen und dem Freenet-Daemon. Es ist unglaublich ineffizient und noch dazu nicht verschlüsselt. Dies ist der Grund für die Empfehlung, den Daemon nur auf dem lokalen Rechner laufen zu lassen: wäre dumm, wenn man aus dem hochsicheren Freenet über eine ungesicherte Internetverbindung die Windows-Sourcen herunterlädt und sich erwischen läßt (ist alles schon vorgekommen) ...

In Perl geht dies denkbar einfach:

```
use Net::FCP;
my $fcp = new Net::FCP;
```

Man kann dem Konstruktor von `Net::FCP` direkt Namen und Port des Freenet-Daemons übergeben. Üblicherweise holt er sich diese aber aus den Environment-Variablen `FREDHOST` und `FREDPORT`, die auf `127.0.0.1` bzw. `8481` defaulten.

Übrigens wird nur eine virtuelle Verbindung aufgebaut. Möchte man sicherstellen, daß tatsächlich ein Daemon erreichbar ist, kann man einen `client_hello`-Request ausführen oder einfach loslegen.

Im Normalfall heißt das also: `no configuration required`.

Hat man ein `Net::FCP`-Objekt, kann man schon alle Requests durchführen, die man machen möchte:

```
my ($meta, $data) = @{$fcp->client_get (
    "freenet:SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3/"
    15
) };
```

Der `client_get`-Request erwartet zwei Argumente: eine Freenet-URI (`freenet:<freenet-schlüssel>`) und eine HTL. Letztere muss man nicht angeben, man sollte es aber, und vor allem sollte man dem Benutzer die Wahl der HTL überlassen. Das dritte, optionale, Argument ist für spezielle Anwendungen: am besten ignorieren.

Als Ergebnis erhält man immer eine Array-Referenz mit den Metadaten und den Daten. Immer. Sollte ein Fehler auftreten wird eine Exception ausgelöst (auf Perl: das Modul `died`). `eval {}` hilft also im Zweifelsfalle, für einfache Anwendungen ist das aber overkill.

Bricht das Programm ab, so sieht das folgendermaßen aus:

```
Net::FCP::Exception<<short_data,reason:unexpected eof or
internal node error>>
```

Oder, wenn die Daten nicht gefunden wurden:

```
Net::FCP::Exception<<data_not_found,>>
```

Letzteres bedeutet übrigens nicht aufgeben, sondern nochmal versuchen, z.B. mit einer höheren HTL. Ein fehlgeschlagener Versuch mit hoher HTL bedeutet immer noch nicht aufgeben. Daemons in der "Nähe" wissen nun, das die Daten verlangt werden. Nach einiger Zeit ist es wahrscheinlich, das die Daten auf einmal in der Nähe liegen: daher nie

aufgeben, nochmal probieren. Niemand hat behauptet, man bekäme die Daten einfach aus dem Freenet wieder heraus...

Die Metadaten sind als Textdokument gespeichert. Das Format dieser Metadaten ist aber relativ "krank", weshalb das Perl-Modul einen Hash mit den gepackten Daten liefert (genaugenommen ein `Net::FCP::Metadata`-Objekt, aber dieses Modul ist noch in Entwicklung).

Die Metadaten und die eigentlichen Nutzdaten kann man ausgeben:

```
use Data::Dumper;
print STDERR Dumper $meta;
print $data;
```

Damit haben wir im wesentlichen den Quelltext für das "Tool" `eg/fetch1`, mit dem man *einen* Freenet-Datenblock herunterladen kann.

Ich benutze es üblicherweise so:

```
eg/fetch1 SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3// >data
```

Das schreibt die Daten in eine Datei und gibt gleichzeitig dessen Metadaten aus.

Für den Key `SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//` ergibt dies:

```
'version' => { 'revision' => '1' },
'document' => [
  {
    'info' => { 'format' => 'image/png' },
    'redirect' => { 'target' => 'freenet:CHK@pMKW...KAwI,7...-25C2w' },
    'name' => 'activelink.png'
  },
  {
    'info' => { 'format' => 'text/plain' },
    'redirect' => { 'target' => 'freenet:CHK@2hew...KAwI,o...21z91w' },
    'name' => 'description.txt'
  },
  {
    'info' => { 'format' => 'text/html' },
    'redirect' => { 'target' => 'freenet:CHK@kME~...QAwI,3...dy40kA' },
    'name' => 'index.html'
  },
  {
    'info' => { 'format' => 'text/html' },
    'redirect' => { 'target' => 'freenet:SSK@00hV...PAgM/fx/4' },
    'name' => '.next'
  }
]
```

```

    },
    {
      'info' => { 'format' => 'text/html' },
      'redirect' => { 'target' => 'freenet:CHK@kME~...QAwI,3...dy40kA' }
    }
  ],
  'raw' => 'Version
Revision=1
[... gekürzt...]
Info.Format=text/html
End
',
};

```

Die eigentlichen Daten sind leer (null Byte gross), das Dokument enthält also nur Metadaten! Man beachte vor allem die teilweise recht tiefe Verschachtelung.

Der Key `$metadata->{raw}` enthält die Metadaten, wie sie vom Freenet kamen. Dies ist notwendig, da man manchmal die Metadaten zum Prüfen in der exakten Form benötigt, wie sie hochgeladen wurden. Ansonsten kann man sie ignorieren.

Der Key `$metadata->{document}` enthält Informationen über das Dokument oder in diesem Fall die Dokumente: Ein Hash pro Dokument. Schaut man sich den Inhalt an (ein Array), so sieht man in jedem Hash ein `name`-Key, der den Namen des Subdokuments angibt (bis auf den letzten, darauf komme ich gleich).

Schaut man sich die Freenet-URIs dieser Freesite an, so sieht man derartige URIs:

```

SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//          # Hauptseite
SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//activelink.png # Icon
SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/3//description.txt # für Spider

```

Wenn diese Dokumente mit `client_get` anfordert, bekommt man immer obiges Dokument. Freenet ignoriert nämlich alles, was hinter dem `//` steht und liefert daher immer das gleiche Dokument aus dem SSK-Bereich. (Darauf verlassen sollte man sich nicht unbedingt, in Zukunft reagiert der Freenet-Daemon vielleicht anders. Als Freetter hat man eben nicht leicht).

Der Teil hinter den beiden Slashes (`//`) ist nun das Unterdokument, das man über den Namen identifizieren kann:

```

$activelink = grep $_->{name} eq "activelink.png",
                @{ $metadata->{document} };

```

Fehlt der `name`-Eintrag, ist es der Eintrag mit dem leeren Namen, in diesem Fall der erste Link, der nichts nach den `//` stehen hat. Der `name`-Key *kann* vorhanden und leer sein:

Freenet überprüft die Daten nicht, daher gibt es durchaus unterschiedliche Auslegungen für das genaue Format.

Die Metadaten sind etwas genauer in dem *hüstel* etwas veralteten *hüstel* *Freenet Explained*-Dokument beschrieben.

Möchte man nun das `activelink.png` herunterladen muss man in `$activelink->{redirect}` nachsehen. Meistens verweist dies auf einen CHK-Schlüssel. Letztere sind sozusagen Allgemeingut: jeder kann sie benutzen, und da sie nicht änderbar sind, kann sie auch niemand fälschen, daher muss man sie nicht signieren.

Auf `$activelink->{info}{format}` sollte man sich übrigens nicht verlassen.

Und sollte kein `redirect`-Key vorhanden sein, so ist das Dokument im mitgelieferten Datenblock. Alles ganz logisch, einfach, und klar, wie man sieht.

Egal, der CHK-Key für `activelink.png` liefert folgendes:

```
'version' => { 'revision' => '1' },
```

Und die Daten stellen ein PNG dar. Naja, nicht gerade üppig, die Metadaten, aber man hätt's ja im Link auf dieses Dokument sehen können. Als Freenetter hat mans nicht leicht.

Aber mit diesen Erklärungen kann man schon einfache Freenet-Spider bauen. Die wichtigsten Tools sind veraltete und spärliche Dokumente wie *Freenet Explained* und Tools wie `Data::Dumper`. "Veraltet" ist übrigens nicht immer schlecht, da viele Dokumente auch alt sind oder sich eh' nicht exakt an den "Standard" halten. Mit der Zeit wird hier sicher eine Besserung eintreten.

Methoden des Freesite-Managements

Oder: "Wenn man Inhalte nicht mehr verändern kann, wie verändert man sie?"

Die offensichtliche Methode ist: "Man macht sie veränderbar". In der Freenet-Gemeinde geistert seit langer Zeit der Mythos des *TUK*-Schlüssels, des *Time Updatable Keys*. Diese Schlüssel tauchen immer auf, wenn jemand über veränderbare Schlüssel spricht. Leider weiss niemand, wie man sie implementieren soll, und ganz persönlich fürchte ich, es wird niemals veränderbare Keys geben.

"Editions" Die nächstliegende Methode nutzt die Eigenschaft von SSKs (genauer: redirects) aus, das man den Namen sofort generieren kann, den Inhalt aber erst später einfügt.

Dies nutzt man aus, indem man *Editionen* herausgibt und durchnummeriert. Jede Edition enthält einen Link auf die nächste. Bis man die nächste Edition einfügt, wird der Link-Inhalt nicht gefunden.

Üblicherweise benutzt man in HTML ein `IMG`-Element mit Link auf ein Bild aus der nächsten Edition. Sieht man das Bild, weiss man, die nächste Edition existiert und kann sich hinklicken.

Freenet Explained benutzt diese Methode. Der Link auf Edition 4 lautet:

```
SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/4//          # Link-Ziel
SSK@00hVDWutibbBMbXmbxNXWOM6YFoPAgM/fx/4//activelink.png # IMG-SRC
```

Beides existierte nicht, als ich diesen Artikel schrieb (bzw. Freenet konnte es nicht finden :). Wenn eine neue Edition herausgegeben werden soll, fügt der Autor einfach einen neuen Datenblock mit vielen Redirects unter obigem SSK-Key ein.

Ein `.next`-Eintrag ist ebenfalls vorhanden: manche Spider folgen diesem und können auf diese Weise immer die aktuelle Edition anzeigen, sofern sie häufig genug scannen.

Editionen sind einfach für den Autor einer Freesite und erfordern keine spezielle Unterstützung. In der Benutzung sind sie allerdings umständlich und häufige Updates sind auch nicht effizient.

Daher hat sich eine zweite Methode entwickelt, die gerade bei häufig geänderten Inhalten besser funktioniert:

“Date Based Redirects” Freesites, die “Date Based Redirects” benutzen (kurz “DBR-Sites”) funktionieren ähnlich wie Editionen-basierte Freesites, nur wird statt einer fortlaufenden Nummer die aktuelle Zeit benutzt.

Die Freesite *The Freenet Help Index* (SSK@rjYFfgPHfolmcStiaoxESFfBXz8PAgM/FreenetHelp//) benutzt diese Methode. Folgende Metadaten wurden dazu hinterlegt:

```
'version' => { 'revision' => '1' },
'document' => [
  { 'date_redirect' => {
    'increment' => '15180',
    'target' => 'SSK@rjYFfgPHfolmcStiaoxESFfBXz8PAgM/FreenetHelp'
  }
},
],
```

Statt einem einfachen `redirect` gibt es jetzt einen `date_redirect`-Eintrag, und darin ein `target` (dürfte bekannt sein) und den Key `increment`. Letzterer darf wie üblich fehlen, man sollte dann 86400 annehmen (ein Tag hat 86400 Sekunden). In diesem Beispiel wird 86400 genommen (Hexadezimal 15180).

Habe ich eigentlich schon erwähnt, daß jede Zahl im FCP-Protokoll oder im Freenet hexadezimal kodiert wird? Also, praktisch immer, außer wenn es mal nicht so ist. Und wozu man einen Offset addiert, ist mir auch schleierhaft. In der freien Wildbahn habe ich sowieso noch keinen gesehen.

Der Link in `target` ist nicht vollständig (unter anderem fehlt hinten ein `//`). Der komplette Link wird generiert, “indem die aktuelle Zeit (POSIX-Zeit, üblicherweise das,

was `time` liefert), auf `increment`-Schritte gerundet und `offset` addiert, als Big-Endian-Hexadezimalzahl nach dem ersten Slash mit folgendem Minuszeichen eingefügt wird.”

Und jetzt in Perl - zum Verstehen:

```
my $doc = $metadata->{document}[0]; # Oder [1] oder ...

my $increment = (hex $doc->{increment}) || 86400;
my $offset     = (hex $doc->{offset})     || 0;
my $target     = $doc->{target}          || die;

my ($head, $tail) = split /\//, $target, 2;

my $NOW = time;
my $time = $NOW - $NOW % $increment + $offset;

my $result = sprintf "%s/%x-%s", $head, $time, $tail;
```

Für “jetzt” (`time() == 1084388445`) liefert der Algorithmus folgenden Link:

```
SSK@rjYFfgPHfolmcStiaoxESFfBXz8PAgM/40a16900-FreenetHelp
```

Und tatsächlich, unter diesem Key findet man wieder ein Dokument mit vielen Redirects.

DBRs haben den Vorteil, “automatisch” aktuell zu sein. Solange man nur Redirects einfügt, ist die Belastung durch neue das Einfügen vieler neuer Daten gering, Freenet kommt damit zurecht und löscht alte DBRs bei Bedarf automatisch (ohne natürlich zu wissen, das sich in den Datenblöcken DBRs befinden, denn die kann ja niemand dekodieren, der nicht den Schlüssel besitzt).

Der Nachteil ist, das man eine aktuelle Uhrzeit braucht um die Site zu finden. Schlimmer noch: die Freesite muss regelmäßig neu eingefügt werden, eben alle `increment` Sekunden.

DBR-Freesites, die nicht mehr maintained werden, verschwinden deshalb fast sofort, solange man nicht einen Zeitpunkt kennt, zu dem sie noch (bzw. schon!) existierte.

All dieses *sollte* eigentlich in einem Metadata-Modul stattfinden. So weit bin ich aber nicht, da mein Hauptziel das effiziente Herunterladen großer Dateien ist.

Womit ich beim Thema wäre.

Effizient herunterladen

Transfers im Freenet zeichnen sich durch zwei Eigenschaften aus: hohe Latenz und vergleichsweise hoher Durchsatz. Dies ist ungewöhnlich, ergibt sich aber aus dem Suchverfahren: die Suche nach einem Datenblock kann sehr lange dauern (sogar einige Male fehlschlagen bevor die Daten eintreffen), sie sind jedoch einfach parallelisierbar.

Wie in Java üblich, erreichte man dies früher durch massive Benutzung von Threads. Der Freenet-Referenz-Daemon ist davon inzwischen abgekommen (zu langsam, zu viel Speicher und zu hohe Komplexität), doch viele Clients haben auch heute noch Voreinstellungen wie ‘Anzahl der Threads pro Download’ und viele sind auch so implementiert.

Da Perls aktuelle Thread-Implementierung ein Prozessmodell simuliert ist der Overhead ebenfalls entsprechend hoch, und man erhält bei Thread-Benutzung viele ihrer Nachteile, jedoch keine nennenswerten Vorteil, außer das alles etwas langsamer läuft und man daher besser zuschauen kann :)

Für `Net::FCP` (oder besser: für mich) kam es deshalb nicht in Frage, Parallelisierung auf Thread-Basis zu implementieren. Das Problem auf den Modul-Benutzer abzuschieben erschien mir auch nicht fair, daher unterstützt `Net::FCP` die Module *Coro*, *Event*, *Glib* oder *Tk*, bzw. deren Event-Steuerung. Man hat also die Wahl, auf welcher Basis man sein Programm aufbaut, und es geht auch ganz ohne Event-System (wie die obigen einfachen Beispiele zeigen).

Transaktionen

`Net::FCP` benutzt für jeden Request eine *Transaktion*. Das ist ein Objekt, das den Zustand und eintreffende Ergebnisse speichert.

Jede Anfrage, (z.B. `client_get`) wird intern in ein solches Transaktionsobjekt verwandelt. Für jede ‘einfache’ Methode wie `client_get` oder `insert_private_key` gibt es eine entsprechende Methode mit `txn_-`Präfix, die statt des Resultats ein solches Objekt liefert.

Transaktionsobjekte besitzen einige Methoden, mit denen sie konfiguriert werden können oder Ergebnisse abgefragt werden können. Die wichtigste ist die `result`-Methode, die auf das Ergebnis wartet und es zurückliefert:

```
# $fcp->client_get wird intern so implementiert:
my $txn = $fcp->txn_client_get (...);
return $tdn->result;
```

Es ist übrigens immer gefahrlos, Transaktionsobjekte zu erzeugen. Etwaige Fehler bei der Durchführung werden immer erst beim Aufruf von `result` und immer als Perl-Exceptions gemeldet.

Um also alle Freenet-Objekte im Array `@download` gleichzeitig anzufordern, reicht folgender Code:

```
map $_->result,
  map $_->txn_client_get ($_),
    @download;
```

Zuerst werden alle URIs auf eine entsprechende Transaktion gemapped, und dann von allen Transaktionen das Resultat angefordert.

Natürlich brauchen manche Zugriffe länger als andere. Wenn Transaktionen früher beendet werden, sollte man gleich eine neue starten, um Freenet am Laufen zu halten.

Dazu benötigt man ein Signal, daß eine Transaktion beendet ist. Dies macht `Net::FCP` mit Hilfe eines Callbacks:

```
my $txn = $fcp->txn_client_get (...)
        ->cb (\&callback);
```

Diesen setzt man mit Hilfe der `cb`-Methode. Alle Methoden, die ein Transaktionsobjekt konfigurieren, liefern das Transaktionsobjekt zurück, man kann also Aufrufketten wie im Beispiel bilden.

Der Callback wird aufgerufen, wenn der Request (erfolgreich oder nicht) beendet wurde.

Nun braucht man allerdings die Hauptschleife des jeweiligen Event-Systems, da man ja nicht mehr auf ein bestimmtes Ergebnis wartet, sondern auf ein beliebiges.

Dazu sollte man `Net::FCP` auf ein bestimmtes Event-System zwingen. Das folgende Beispiel tut dies und lädt die Kommandozeileargumente parallel herunter:

```
use Net::FCP qw(event=Event); # oder event=Glib, oder...

my $fcp = new Net::FCP;
$fcp->txn_client_get ($_)->cb (\&finished) for @ARGV;

Event::loop; # hier passiert, loop ist die Hauptschleife von Event
```

Die Transaktionsobjekte werden übrigens nicht gespeichert, denn sie werden dem Callback übergeben, und nur dort werden sie gebraucht. Der Callback könnte so aussehen:

```
sub finished {
    my ($txn) = @_;
    my ($meta, $data) = @{$txn->result };
    # tue etwas
}
```

Oder, mit Fehlerprüfung:

```
sub finished {
    my ($txn) = @_;
    my ($meta, $data) = eval { @{$txn->result } };
    if ($?) {
```

```

        warn "fehler: $@, wird einfach ignoriert :)"
    } else {
        warn "Lo and Behold! We got soemthing!";
        # tue etwas
    }
}

```

Fortschritt ist nicht aufzuhalten

Da Zugriffe recht lange dauern können, schickt Freenet regelmäßig Fortschrittsreports. Man kann sich das vorstellen wie `warn` und `die: warn` liefert wichtige Informationen, die einen Fehler.

Die Fortschrittsanzeigen werden normalerweise nicht von `Net::FCP` ausgegeben (wäre irgendwie kontraproduktiv in einem GUI-Programm), man kann aber einen Callback installieren:

```
my $fcp = Net::FCP progress => \&progress;
```

Ein solcher Progress-Callback könnte so aussehen:

```

sub progress_cb {
    my ($self, $txn, $type, $attr) = @_;

    warn "progress<$txn,$type," . (join ":", %$attr) . ">\n";
}

```

Möchte man Informationen für den Callback bereitstellen, so bietet sich die `userdata`-Methode von Transaktionsobjekten an.

Hier ist ein kompletter parallelisierender Client mit Fortschrittsanzeige:

```
my $fcp = Net::FCP progress => \&progress;
```

Ein solcher Progress-Callback könnte so aussehen:

```

use Net::FCP qw(event=Event); # oder event=Glib, oder...

my $fcp = new Net::FCP progress => \&progress_cb;

$fcp->txn_client_get ($_)->cb (\&finished) for @ARGV;

Event::loop; # hier passiert, loop ist die Hauptschleife von Event

```

```
sub finished {
    my ($txn) = @_;
    my ($meta, $data) = eval { @{$txn->result} };
    if ($?) {
        warn "$txn: fehler: $?, wird einfach ignoriert :)\n"
    } else {
        warn "$txn: Lo and Behold! We got something!\n";
        # tue etwas
    }
}

sub progress_cb {
    my ($self, $txn, $type, $attr) = @_;

    warn "$txn: progress<$txn,$type," . (join ":", %$attr) . ">\n";
}
```

Startet man ihn mit einigen gültigen und ungültigen Freenet-URIs, bekommt man vielleicht folgende Ausgabe:

```
Net::FCP::Txn::ClientGet=HASH(0x81e0f64): ...
... fehler: Net::FCP::Exception<<uri_error, ...
... reason:Unspecified document name>>, wird einfach ignoriert :)
Net::FCP::Txn::ClientGet=HASH(0x81e7254): ...
... fehler: Net::FCP::Exception<<uri_error,reason: ...
... Unknown keytype>>, wird einfach ignoriert :)
Net::FCP::Txn::ClientGet=HASH(0x81e0c4c): ...
... progress<Net::FCP::Txn::ClientGet=HASH(0x81e0c4c), ...
... data_found,data_length:74:metadata_length:74>
Net::FCP::Txn::ClientGet=HASH(0x81e0c4c): ...
... progress<Net::FCP::Txn::ClientGet=HASH(0x81e0c4c), ...
... data,chunk:116:total:116:received:116>
Net::FCP::Txn::ClientGet=HASH(0x81e0c4c): ...
... Lo and Behold! We got something!
... usw.
```

Benutzt man das Coro-Modul, so kann man Callback-basierte und Thread- (bzw. Coroutinen-) basierte Programmierung mischen:

```
use Net::FCP qw(event=Coro);

my $fcp = new Net::FCP;
```

```

# Achtung. Pseudocode :)
for my $url (@urls) {
    async {
        my ($meta, $data) = @{ $fcp->client_get ($url) };

        if ($meta ist ein splitfile) {
            for (@alle_splitfile_blöcke) {
                $fcp->txn_client_get (...)
                    ->cb (\&save_splitfile_block);
            }
        } else {
            # direkt speichern
        }
    };
}

```

Splitfiles

Das Them Splitfiles möchte ich hier noch erwähnen, aber nicht ausführlich besprechen.

Da die Datenblöcke im Freenet (zur Zeit) nicht größer als 1MB sein können, müssen größere Dateien (Filme z.B.) aufgeteilt werden. Wenn man viele Datenblöcke für eine Datei braucht, erhöht sich die Wahrscheinlichkeit dafür, daß ein Block mal fehlt oder nicht gefunden werden kann, stark auf einen Wert, der das Herunterladen großer Dateien unmöglich macht.

Daher wird mit Fehlerkorrekturcodes die Anzahl der Blöcke um 50% erhöht. Metadaten für ein Splitfile sehen so aus:

```

'version' => { 'revision' => '1' },
'document' => [ {
    'info' => {
        'checksum' => '3bab309132f812cb2a281357b0dc1036920672e2',
        'format' => 'video/mpeg',
        'description' => 'Union FEC v1.2 file inserted by Fuqid'
    },
    'split_file' => {
        'algo_name' => 'UnionFEC_a_1_2',
        'check_block_count' => '3f',
        'block_count' => '7f',
        'check_block' => {
            '33' => 'freenet:CHK@eTNQ...3CBzTSsI~oPn6EUAWI,I4X...nyGoIRZBLg',
            '32' => 'freenet:CHK@Ee1h...Kg3WZ9OKXqwQZMUAWI,ppn...k1z3Sempqw',

```

```

    '1e' => 'freenet:CHK@5gBB...fj027k0tU0hZgkUAwI,0kf...wvtX~wfVNA',
    [viele Zeilen fehlen hier :]
    '3d' => 'freenet:CHK@iIT1...y0ToFsyytV-roUAwI,9u3...-eY70nag1g',
    '5' => 'freenet:CHK@EeVsn...CNBNVzvTTQ110UAwI,q6e...0h-cc79WBA'
  },
  'block' => {
    '33' => 'freenet:CHK@j522...h2ZK1D1q4UDnFgUAwI,Xwo...XFwI5f9iyw',
    '32' => 'freenet:CHK@VU5v...y-Dnk8Sct9ZKbYUAwI,iX~...3YkOrmpM9w',
    '5d' => 'freenet:CHK@-wbX...uotK~uZbJp14zsUAwI,-bg...YvbwQeYtyA',
    [viele Zeilen fehlen hier :]
    '43' => 'freenet:CHK@zMXc...ee1cftvdK4~QukUAwI,o99...adXUXGjrlg',
    '5' => 'freenet:CHK@1ZDiv...YTKQbDq1yCPWAUAwI,B9f...f0x~PTZkAA',
    '3d' => 'freenet:CHK@ZGxa...qQIf6rbr-g8btMUAwI,f0x...mmxoJRzaLw'
  },
  'size' => '7e8e804'
}
} ],
}

```

Die Datenblöcke und die zusätzlichen Korrekturblöcke sind getrennt aufgeführt. Man braucht nur so viele Blöcke insgesamt, wie die Datei groß ist. Den Rest kann man aus den heruntergeladenen erzeugen.

Wie genau die Daten auf die Blöcker verteilt werden unterliegt einem Algorithmus und ist nicht in den Metadaten gespeichert. Der Grund ist, daß jedes Programm denselben Algorithmus benutzen muss, und Dateien daher immer gleich auf die Blöcke verteilt werden, so daß Mehrfacheinfügen ins Freenet keine Duplikate erstellt.

Den genauen Algorithmus kann man in der Datei `bin/fmd`, dem *Freenet Mass Downloader* in der Funktion `state_splitfile` nachlesen. Der Code ist weder besonders klar, noch besonders kurz, noch besonders hübsch. Auch dieser Teil wird irgendwann einmal in ein Modul fließen, damit er anderen zugänglicher wird.

Interessant ist noch, daß man CHK-Inhalte, die aus dem Freenet heruntergeladen werden, einfach auf Konsistenz prüfen kann: sie enthalten schließlich den SHA1-Hash der Daten:

```

use Net::FCP::Util;

my ($meta, $data) = @{ $fcp->client_get ($uri) };

# Extrahiere den Hash aus der URI
my $k1 = Net::FCP::Util::extract_chk_hash $uri;
# Berechne den Hash aus den Daten
my $k2 = Net::FCP::Util::generate_chk_hask "$meta->{raw}$data";

```

```
# Idealerweise stimmen beide überein...
$k1 eq $k2 or die "Key/Content mismatch!";
```

Zwar ist der Freenet-Daemon angeblich perfekt, aber früher passierte es tatsächlich oft, daß man korrupte Daten geliefert bekam. Retryen half. Man hat es eben nicht so leicht...

Heute ist das zwar nicht mehr der Fall (...), aber prüfen kostet fast nichts.

Publizieren

Das Freenet lebt von vielen Dingen: Den Entwicklern, den Benutzern die ihr Interesse bekunden, den Regierungen, die die Meinungsäußerung einschränken - aber vor allem dadurch, daß es sinnvolle Inhalte darin gibt.

Ich bin mir nicht so sicher, *ob* es so viele sinnvolle Inhalte gibt, aber noch gab es ja auch keine 1.0-Release...

Nun ja, solange man nichts ins Netz stellt, darf man auch nicht klagen.

Das kann man ändern, indem man so langweilige Dinge wie den *Freenet Insertion Wizard* oder das beliebte *Fuqid* benutzt (was ich noch nie getan habe). Aber die benutzen kein Perl und sind damit langweilig.

Mit `Net::FCP` geht es erstmal sehr einfach:

```
$fcp->client_put ($uri, $metadata, $data, $ttl, $remove_local);
```

(Auch hier gibt es natürlich wieder die entsprechende `txn_`-Variante).

Einfügen von Inhalten geht also fast genauso wie Auslesen. `$uri` ist eine Freenet-“URIC” (mit oder ohne `freenet:`), wie man zu dieser kommt gleich mehr.

`$metadata` kann entweder ein Freenet-Metadata-String sein oder ein Hash der gleichen Form wie man ihn zurückgeliefert bekommt. `$data` sind natürlich die eigentlichen Daten und `$ttl` sind die Hops-To-Live, die der Key-Insert haben soll. Höhere Werte (bis hin zu 25) speichern die Daten dauerhafter, brauchen aber auch entsprechend länger.

`$remove_local`, auf 1 gesetzt, sorgt dafür, daß der lokale Netzknoten die Daten *nicht* cached. Warum man das tun sollte, ist einfach erklärt: wenn Freenet beim Einfügen die Daten schon im Netz vorfindet, wird der Einfügevorgang abgebrochen. Möchte man die Daten regelmäßig “auffrischen” dann sorgt diese Option dafür, daß die Daten den Knoten auf jeden Fall verlassen und erhöhen so die Chance auf Verbreitung.

Nun die Frage wie man zu einer URI kommt. Möchte man einen CHK-Schlüssel einfügen, so setzt man die URI einfach auf den String `CHK@`. Nach dem Einfügen erhält man die eigentliche URI als Resultat:

```
my $attr = $fcp->client_put ('CHK@', "", "Daten\n", 20);
print "eingefügte URI: $attr->{uri}\n";
```

Als Ergebnis dieses `client_put`-Requests erhielt ich:

```
freenet:CHK@MT~LuxKHH8fugJehkcgp239h7C4KAwI,378rJzHVZbsQbd7VGMHxSg
```

Beim Einfügen ins Freenet kann es auch einen *Key Collision*-Fehler geben, und zwar genau dann, wenn Freenet die Daten schon findet, die man einfügen wollte.

`Net::FCP` behandelt diesen Fehler allerdings genauso wie ein erfolgreicher Einfügevorgang, lediglich im Hash, den es zurückliefert, wird das `key_collision`-Element auf 1 gesetzt. Wenn die Daten schon vorhanden sind, ist das Ziel schließlich auch erreicht.

Häufig möchte man den `CHK`-Schlüssel aber schon im voraus Wissen. Dazu kann man entweder seinen freundlichen Freenet-Daemon fragen:

```
my $key = $fcp->generate_chk ($metadata, $data);
```

Oder man verwendet das `Net::FCP::Key::CHK`-Modul, das schneller ist und keinen Serverzugang benötigt:

```
use Net::FCP::Key;

my $key = (new_from_data Net::FCP::Key::CHK $metadata, $data)->chk;
```

SSK-Schlüssel generieren

SSK-Schlüssel bestehen immer aus zwei Teilen, dem öffentlichen (`public_key`) und dem privaten (`private_key`), die zusammen ein Paar bilden. Neuerdings noch aus einem dritten, dem `crypto_key`, der es Brute-Force-Attacken noch schwieriger machen soll.

Solch ein Schlüsselpaar kann man mit einem Aufruf von `generate_svk_pair` (auch hierfür gibt es eine `txn_`-Variante) erzeugen. Das "svk" ist übrigens kein Schreibfehler: SSK-Schlüssel sind ein Spezialfall von SVK-Schlüsseln, die man aber als normalsterblicher nirgendwo direkt antrifft.

```
my ($public, $private, $crypto) = @{ $fcp->generate_svk_pair };
```

Daraus kann man beliebig viele SSK-Schlüssel erzeugen, indem man die drei Teile wie folgt zusammensetzt:

```
# Öffentlicher Schlüssel:

my $name = "..."; # beliebig

my $get = "SSK\@${public}PAgM,$crypto/$name";
my $put = "SSK\@$private,$crypto/$name";
```

Den mit `$put` bezeichneten Schlüssel muss man zum Einfügen verwenden. Mit dem mit `$get` bezeichneten Schlüssel kann man diese Daten wieder holen. Letzterer kann bedenkenlos veröffentlicht werden.

Das SSK-Management kann vom `Net::FCP::Key::SSK`-Modul übernommen werden.

Eine Freesite erzeugen

Um eine Freesite zu erzeugen, sollte man sich zuerst einen dauerhaften SSK-Schlüssel besorgen:

```
# perl -MNet::FCP -MNet::FCP::Key::SSK \  
-e 'Net::FCP::Key::SSK->new_from_fcp (Net::FCP->new) \  
->save ($ARGV[0])' mykey
```

Das erzeugt einen neuen SSK-Schlüssel und speichert ihn gleich in der Datei `mykey`. Dieser SSK-Schlüssel wird die Basis der Freesite.

Für das Einfügen schreibt man sich am besten ein Skript, da man dies regelmäßig wiederholen sollte. Zuerst ein paar Definitionen:

```
my $edition = 1;  
my $keyfile = "mykey";  
my $html = 10;
```

Das Skript legt eine editionenbasierte Freesite an und benutzt als Basis den eben erzeugten SSK-Schlüssel. Der Hops-To-Live-Wert von 10 ist etwas klein, aber so dauert das Einfügen vielleicht nur ein paar Minuten... In der Praxis sollte man 20-25 vorziehen, oder das Einfügen regelmäßig wiederholen.

Als nächstes benötigen wir ein paar Module:

```
use Net::FCP qw(event=Event);  
use Net::FCP::Metadata;  
use Net::FCP::Key::SSK;
```

Generell sollte man in einem Perl-Programm den verwendeten Event-Mechanismus fest angeben. In Freenet-Modulen sollte man dies dagegen nicht tun, um dem Benutzer die Wahl zu überlassen. Stattdessen sollte man in Modulen nur Transaktionen benutzen und diese dem Benutzer zur Verfügung stellen.

Als nächstes werden ein paar notwendige Objekte erzeugt:

```
my $fcp = new Net::FCP;  
my $ssk = new_from_file Net::FCP::Key::SSK $keyfile;  
my $manifest = new Net::FCP::Metadata;
```

Die Metadaten für die Hauptseite (`$manifest`) werden auch "Manifest" genannt, da man dort häufig nur Verweise auf die jeweiligen Unterseiten ablegt, also sowas wie der Grundstein einer Freesite. Das `Net::FCP::Metadata`-Modul bietet zur Zeit nur spärliche Funktionalität, für unsere Zwecke reicht das jedoch aus.

Nun noch zwei Hilfsfunktionen. Zuerst `add_key`:

```
sub add_key {
    my ($name, $key, $meta, $data) = @_;

    # Asynchrones Einfügen
    $fcp->txn_client_put ($chk, $meta, $data, $htl, 1)
        ->cb (sub {
            eval {
                my $attr = $_[0]->result;
                printf "Insert of '%s' (%d bytes) ok.\n", $name, length $data;
            } or warn "Error while inserting '$name': $@";
        });
}
```

`add_key` startet eine `client_put`-Transaktion, ohne auf das Ende zu warten. Im Call-back wird das Resultat angezeigt, oder, wenn ein Fehler auftrat, eben dieser. Das Skript sollte so angelegt sein, das man es einfach nochmal starten kann, wenn einige Schlüssel nicht eingefügt werden konnten.

Nun noch `add_file`:

```
# Hilfsfunktion zum Einfügen von Dateien
sub add_file {
    my ($name, $path, $contenttype) = @_;

    # lies die Datei ein
    my $data = do {
        local $/;
        open my $fh, "<", $path
            or die "$path: $!";
        <$fh>
    };

    my $chk = $fcp->generate_chk ("", $data); # keine metadaten
    $manifest->add_redirect ($name => $chk, format => $contenttype);
    add_key $name, $chk, "", $data;
}
```

`add_file` lädt eine externe Datei (< 1MB) und berechnet deren CHK-Schlüssel. Diesen Schlüssel fügt es als Verweis unter dem Namen `$name` in das Manifest ein. Als letztes benutzt es `add_key`, um die Datei einzufügen.

Das Vorausberechnen des CHK-Schlüssels macht es möglich, das Manifest paralell zu den anderen Schlüsseln einzufügen. Würde man auf den Einfügevorgang warten, müsste man das Manifest am Ende getrennt hochladen.

Nun ans Zusammenbauen. Zuerst werden (unsichtbare) Verweise auf die vorherige bzw. die nächste Edition eingefügt:

```
$manifest->add_redirect (".prev" => $ssk->gen_pub ($edition - 1))
    if $edition > 1;
$manifest->add_redirect (".next" => $ssk->gen_pub ($edition + 1));
```

Dies nutzt zum einen die Eigenschaft von SSKs, einfach vorausberechnet werden zu können, als auch die Methode `gen_pub`, die den öffentlichen SSK-Schlüssel mit optionalem Suffix generiert.

Dadurch können Benutzer auch “per Hand” eine neuere oder ältere Edition suchen, indem sie einfach die Zahl (`.../2//index.html`) vor dem `//` ändern.

Als nächstes folgt der eigentliche Inhalt der Freesite, eine HTML-Seite, ein Bild und der Quellcode des Skriptes, der sie generiert hat:

```
add_file "activelink.png" => "logo.png", "image/png";
add_file "" => "index.html", "text/html";
add_file "index.html" => "index.html", "text/html";
add_file "freesite" => "freesite", "text/plain";
```

Die Datei `index.html` wird zweimal hinzugefügt, einmal unter dem Namen `""` und ein zweitesmal unter dem Namen `index.html`. Dadurch kann man erreichen, was viele HTTP-Server ebenfalls machen, nämlich eine URI, die auf ein Verzeichnis zeigt, auf eine Datei darin zu verweisen. Das die Datei zweimal eingefügt wird ist kein Problem, da der CHK-Schlüssel derselbe ist (und Mehrfacheinfügen kein Fehler).

Nachdem die Dateien hinzugefügt (und vor allem ins Manifest eingetragen) wurden, kann man das Manifest einfügen:

```
my $site = $ssk->gen_pub ($edition);
add_key "manifest", $site, $manifest, "";
```

Das Manifest wird. Damit man hinterher auch weiss, was man da eingefügt hat, sollte man den Hauptschlüssel auch ausgeben:

```
print "site generated as $site//\n";
```

Und da die ganzen Einfügevorgänge ja nur gestartet wurden, muss man auf deren Ende warten:

```
Event::loop;
```

Solange `Event` noch aktive `Watcher` sieht, bleibt es in der Event-Schleife. Wenn es keinen aktiven `Watcher` mehr gibt, kehrt `loop` automatisch zurück und das Programm ist beendet.

Die Ausgabe sieht dann z.B. so aus:

```
site generated as SSK@wg...TBRylfnDlUBfjETgPAGM,Jr...UBDPfw1U01g/1//
Insert of 'freesite' (1688 bytes) ok.
Insert of 'activelink.png' (1346 bytes) ok.
Insert of 'index.html' (200 bytes) ok.
Insert of '' (200 bytes) ok.
Insert of 'manifest' (0 bytes) ok.
```

Wenn man einen Fehler gemacht hat, kann man einfach einen neuen SSK-Schlüssel generieren. Das heißt, solange man seine Site nicht schon veröffentlicht hat.

Die Alternative wäre z.B. ein täglicher Date-Based-Redirect, d.h. tägliches komplettes Neueinfügen. Da bügeln sich Fehler schnell von selbst aus, man muss sein Skript dann natürlich über Tage oder Monate regelmäßig starten.

Eine offensichtliche Verbesserung wäre es, HTML-Dateien abzuändern, so daß man Links auf die nächste Revision nicht hardcodieren muss sondern etwa als `%%NEXT%%//activelink.png` schreiben könnte.

Nun ja, ich hoffe, es machen sich nun einige Leser auf, um eigene Freesites zu erstellen oder andere Anwendungen für das Freenet zu erstellen.

Verweise

<http://www.freenetproject.org/>

Die Hauptseite des Freenet-Projekts. Hier gibt es auch White-Papers, etwas Dokumentation und Mailinglisten(-archive).

<http://www.freenethelp.org/>

Hier findet der gestresste Freenetter Hilfe beim Einrichten und Betrieb seines Knotens. Vielleicht.