

# Statemaschine

für universellere Nutzung von Modulen

Steffen Ullrich, genua  
<http://noxxi.de/talks.html>



```
my $request = Mail::SPF::Request->new(  
  identity    => 'foo@example.com',  
  ip_address => '192.168.0.1',  
  ...  
);
```

```
my $result = Mail::SPF::Server->new  
->process($request);
```

DNS Abfragen: **Warten** bis fertig



- .. Blockiert für DNS-Anfragen
- .. Parallelisierung nur durch Threads  
Integration in Event-Loops nicht möglich
- .. genugate Relays sind aber alle event-driven



- Enge Verzahnung von Protokolllogik und I/O  
Nachnutzung der Protokolllogik unmöglich
- Eigene DNS-Logik (Quellen, Caching...) mit  
Net::DNS::Resolver kompatiblen Objekt
- DNS Anfragen nur sequentiell, selbst wenn  
eigener Resolver



```
my $spf = Mail::SPF::Iterator->new(
    $ip, $mailfrom, $helo, $myname);

my ($result,@q) = $spf->next;
while (!$result and my $q = shift(@q)) {
    ... resolve DNS query $q ...
    ($result,@q) =
        $spf->next($dns_answer);
}
```

old state + DNS reply → new state



- Push vs. Pull
  - Output: DNS-Anfragen | Endresultat
  - Input: Antworten auf DNS-Anfragen
- Integration in beliebige Event-Loops  
Blockierend arbeiten weiterhin möglich



- **Volle Kontrolle über DNS-Lookups**  
Timeout, Caching, Quellen, Anzahl, Gesamtzeit ...  
Macht auch Testen einfacher
- **Parallele DNS-Lookups möglich**  
next liefert alle weiteren Anfragen (SPF, TXT, ...)  
Höhere Geschwindigkeit durch Parallelisierung



- Mail::DKIM::Iterator
  - Input:  
DNS-Antworten + Mailchunks
  - Output:  
DNS-Anfragen + „mehr Daten“  
bzw. Endresultat
- Mail::DMARC::Iterator  
kombiniert SPF + DKIM





- Diverse blockierende Bibliotheken  
LWP::UserAgent, HTTP::Tiny, Furl...
- Diverse asynchrone mit eigenem Eventloop  
AnyEvent::HTTP, Mojo::UserAgent
- Verarbeiten von Datenströmen (PCAP...)  
Net::Inspect::L7::HTTP



- Begrenzte Kontrolle des Verhaltens  
über Parameter, Callbacks ...
- Unterschiedliche Features  
Datenkompression ...
- Unterschiedliche Korrektheit  
Chunked Encoding, Keep-Alive ...



- **NEEDMORE**  
brauche mehr Daten
- **HEADER**  
Request/Response-Header eingelesen
- **BODY**  
Payload Daten, evtl. decodiert
- **DONE**  
Ende des Body
- **ERROR**  
fataler Protokollfehler
- **WARN**  
nicht-fatales Problem, z.B. kaputte Headerzeile



```
print $to_server $http_request;
my $state = my::HTTParser->new('response');
my @result = $state->next;
while (@result) {
    my $what = shift(@result);
    if ($what == HTP_NEEDMORE) {
        sysread($cl, my $data, 8192);
        @result = $state->next($data);
    } elsif ($what == HTP_HEADER) {
        my $hdr = shift(@result);
    } elsif ($what == HTP_BODY) {
        my $decoded_payload = shift(@result);
    } elsif ($what == HTP_DONE) {
        last;
    }
    ... HTP_ERROR, HTP_WARN
}
```



- .. Push statt Pull
  - .. Input:  
Chunks von Daten
  - .. Output:  
Done | Need more  
Typisierte Datenchunks  
Informationen zu Problemen



- Unabhängig von Event-Loop  
Auch blockierend verwendbar
- Trennung Protokolllogik und I/O
- Auch ohne Sockets verwendbar, z.B. zur Analyse von mitgeschnittenen Daten
- Volle Kontrolle über applikationsspezifische Abbruchbedingungen, wie Timeouts, Bandbreitenlimits, Größe Response ...



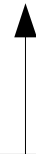
```
while (@result) {  
    ...  
} elsif ($what == HTP_HEADER) {  
    my $hdr = shift(@result);  
    # use raw content not decoded  
    $state->decoder(undef);  
} elsif ($what == HTP_BODY) {  
    my $raw_payload = shift(@result);
```



Höhere Flexibilität  
Direkter Einfluss auf Decoding



```
my $payload = '';  
while (@result) {  
    ...  
} elsif ($what == HTP_HEADER) {  
    my $hdr = shift(@result);  
} elsif ($what == HTP_BODY) {  
    $payload .= shift(@result);  
    last if length($payload) > $MAX;  
}
```



Größenkontrolle ohne spezielle  
Callbacks oder Parameter





- Kann transferiert werden (IMAP, SMTP) oder lokal in unterschiedlichen Formaten vorliegen (Mbox, Maildir, Outlook PSF...)
- Besteht meist aus vielen unterschiedlichen Teilen: Plain-Text, HTML-Text mit Bildern, Attachments, PGP-Signatur ...
- Teile sind kodiert mit quoted-printable, base64, ...



## MIME – typische Mail als Text

```
From: foo@example.com
To: bar@example.com
Subject: foobar
Content-type: multipart/mixed;boundary=foo

--foo
Content-type: multipart/alternative; boundary=bar

--bar
Content-type: text/plain; charset=iso-8859-15
Content-Transfer-Encoding: quoted-printable

text =C4rger ...
--bar
Content-type: text/html
Content-Transfer-Encoding: base64

CjwhZG9jdHlwZSBodG1sPiA...
--bar--
...
```



- Mail mit Text und PDF-Attachments
- Text als Plain und HTML
- HTML mit eingebettetem Firmenlogo

```
multipart/mixed
|- multipart/alternative
|  |- text/plain (quoted-printable)
|  |- multipart/mixed
|     |- text/html (base64)
|     |- image/png (base64)
|- application/pdf (base64)
```



- Vielfalt unterschiedlicher Aufgaben
  - Attachments auf Malware überprüfen (extrahieren, decodieren)
  - Attachment für Transport in IMAP bereitstellen (extrahieren, nicht decodieren)
  - Filenamen Attachment ermitteln um z.B. \*.exe zu blockieren (nur Analyse Header)
  - Text-Teil optimal anzeigen, z.B. nur Plain oder HTML mit Bildern
  - ...



- Existieren diverse Parser für Mail  
Mail::Parser, Email::MIME
- Wollen alle volle Kontrolle über Filehandle  
Extraktion aus Mailbox oder anderen  
Quellen nur über Umwege
- Blockierendes Lesen  
Parallelisierung nur über Threads  
Keine eventbasierte Verarbeitung



- .. Begrenzte Einflussnahme, ob und wie extrahierte Daten abgelegt werden
- .. Analyse der MIME-Parts erst nach kompletter Verarbeitung möglich, kein Streaming
- .. Alles wird extrahiert und decodiert, nicht nur die benötigten Teile



- Trennung Protokolllogik und I/O  
Streaming Parser mit Push der Daten  
Datenquelle egal: Datenbank, MBox, Socket...
- Spamanalyse und Virenschanning  
ermöglichen noch bevor die Mail komplett  
gelesen ist
- Optimierungen, Parametrisierung on-the-fly
  - Decoding nach Bedarf
  - Überspringen von MIME-Parts



- NEEDMORE  
brauche mehr Daten
- SINGLEPART\_HEADER, MULTIPART\_HEADER  
Header des Parts eingelesen
- BODY  
Payload Daten, evtl. decodiert
- MULTIPART\_DONE, DONE  
Ende einer Multipart-Section bzw. der Mail
- ERROR, WARN





## Nutzung im Code (PoC)

```
my $parser = my::MIME::Parser->new;
my @result = $parser->next;
while (@result) {
    my ($what,$arg) = splice(@result,0,2);
    if ($what == MIME_NEEDMORE) {
        sysread($fh,my $buf,8192);
        @result = $parser->next($buf);
    } elsif ($what == MIME_SINGLEPART_HEADER) {
        ...
    } elsif ($what == MIME_MULTIPART_HEADER) {
        ...
    } elsif ($what == MIME_BODY) {
        ...
    } elsif ($what == MIME_DONE) {
        ...
    }
}
```



- .. Push statt Pull
  - .. Input:  
Chunks von Daten
  - .. Output:  
Done | Need more  
Typisierte Datenchunks  
Informationen zu Problemen



multipart/mixed

- [0] MULTIPART\_HEADER
- multipart/alternative
  - [1] MULTIPART\_HEADER
  - text/plain (quoted-printable)
    - [2] SINGLEPART\_HEADER
    - [2] BODY\*
  - text/html (base64)
    - [2] SINGLEPART\_HEADER
    - [2] BODY\*
    - [1] MULTIPART\_DONE
- application/pdf (base64)
  - [1] SINGLEPART\_HEADER
  - [1] BODY\*
  - [0] MULTIPART\_DONE
  - [\*] DONE



- Unabhängig von Event-Loop  
Auch blockierend verwendbar
- Trennung Protokolllogik und I/O
- Unabhängig von Datenquelle  
EML-Datei  
einzelne Mails aus MBox  
Netzwerkdatenstrom  
PCAP-Daten  
...



```
..
} elseif ($what == MIME_SINGLERPART_HEADER) {
  if ($arg->get('content-disposition')
    !~m{^attachment\b}i) {
    $parser->skip_part;
  } else {
    .. open $attach_fh ...
  }
} elseif ($what == MIME_BODY) {
  print $attach_fh $arg;
```



- Streaming Generator mit Push der Daten
- Konstruktion neuer Mails ohne Zwischenspeichern oder großen Speicherverbrauch
- Zusammen mit streaming Parser selektive Veränderung von Mail, z.B. Ersetzen von Attachments



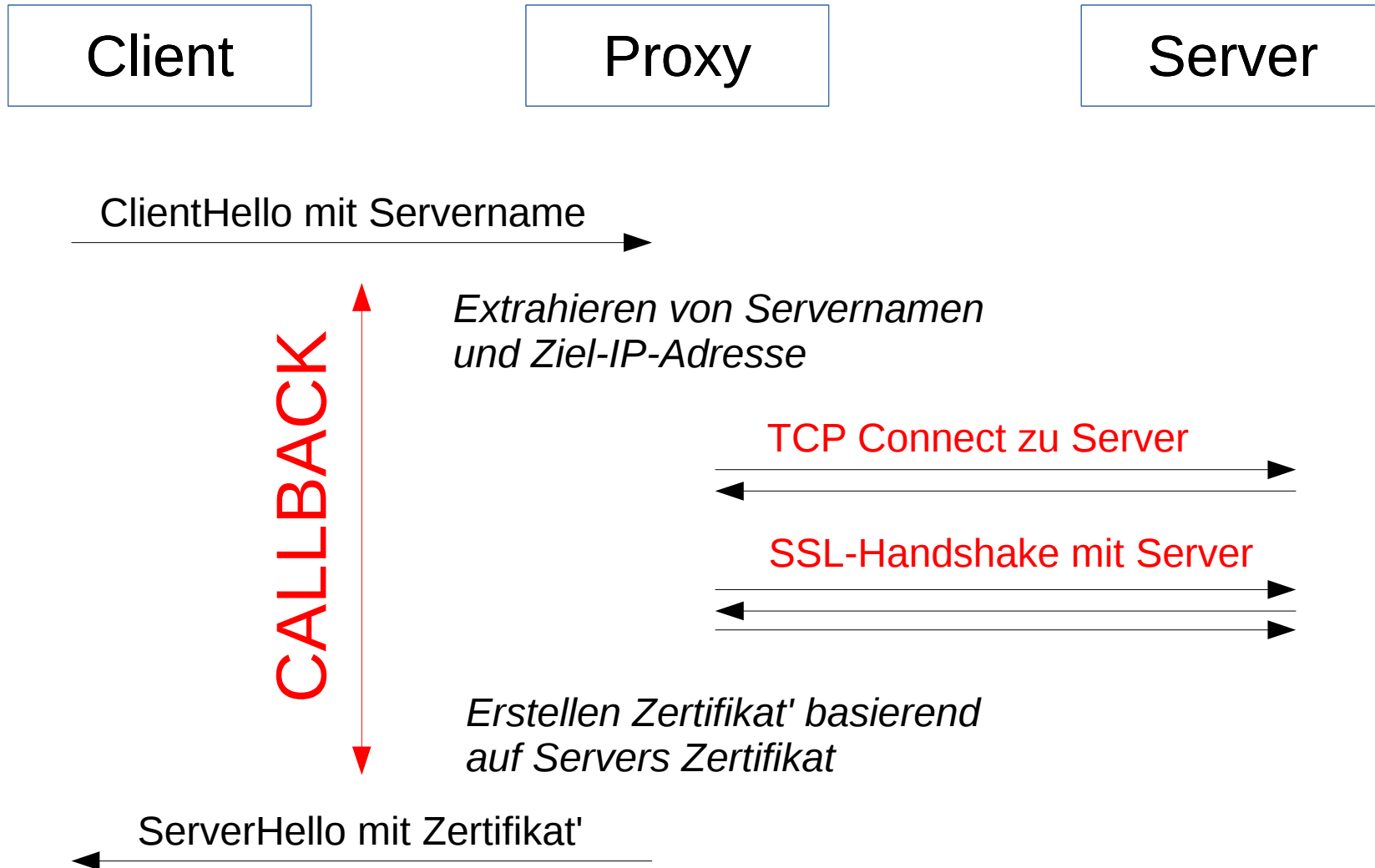
- HTML  
HTML::Parser ist callback-driven
- XML – alles callback driven  
XML::LibXML::Iterator  
XML::Parser  
XML::SAX::\*
- Callbacks evtl. blockierend, z.B. zum  
Auflösen externer Entities



- OpenSSL ist fast eine Statemaschine  
Push über BIO Funktionen
- Einige wichtige Sachen jedoch über  
Callbacks realisiert
  - Behandlung von SNI
  - Validierung von Zertifikaten
- Dumm, wenn diese Callbacks blockieren







- Statemaschine ist mächtiges Konzept
  - Push vs. Pull:  
Einsatz in blockierenden und eventbasierten Situationen
  - Trennung Protokollogik und I/O:  
Einsatz außerhalb traditioneller Situationen, z.B. zur Analyse von Datenmitschnitten
  - Höhere Flexibilität ohne spezielle Callbacks und Parametrisierungen
  - einfache Serialisierung des States für Persistenz

