# BEEP - The Blocks Extensible Exchange Protocol Core

Fangming Xu

**Abstract**

Bei der Diskussion über die Wiederverwendbarkeit des HTTP als Lösung für die Verbindung zwischen Anwendungsprogrammen ist ein neues Protokoll BEEP – Block Extensible Exchange Protocol - entstanden. BEEP ist schon durch die IETF - Internet Engineering Task Force – standardisiert. BEEP verwendet XML und hat die identischen Einflusskräfte an den Internet-Protokollen, wie an den Dokumenten- und Datenverarbeitungen. BEEP kann auf die verschiedene Transportdienste abgebildet werden, z.B. auf TCP. Durch Nutzen von BEEP kann HTTP entlastet werden, um die Effizienz und Effektivität der Netzwerkanwendungen zu realisieren. In dieser Arbeit werden das Konzept und das TCP-Transportmapping nach RFC 3080 [1] und RFC 3081 [2], sowie die Anwendbarkeit von BEEP vorgestellt.

# 1 Einleitung

BEEP heißt Block Extensible Exchange Protocol. Es ist ein neues, von der IETF standardisiertes Rahmenwerk für Internet-Protokolle. BEEP stellt einige gemeinsame Funktionalitäten zur Verfügung, um Entwürfe neuer Anwendungsprotokolle zu unterstützen [3]. In der Vergangenheit, falls man eine Netzwerkanwendung entwickeln wollte und dabei die Instanzen des Anwendungsprogramms durch TCP/IP miteinander kommunizieren lassen wollte, musste man vor der Überlegung nach der Anwendungslogik zuerst die folgenden Probleme lösen: Verbindungsmechanismus, Authentisierung, Senden/Empfangen der Nachrichten und Behandlung von Fehlern oder Störungen. Die Lösung solcher Probleme benötigt mehr Zeit als die Überlegung des Anwendungsprogramms selbst. Das ist auch die Aufgabe von BEEP. Es implementiert alle grundlegenden Aspekten für die Erschaffung eines neuen Protokolls, so dass die Designer sich nicht mehr um die vorher genannten Probleme kümmern müssen [4].

Der primäre Erfinder von BEEP ist Marshall Rose. Er hat mehr als 60 RFCs geschrieben. In RFC 3080 [1] hat er die Grundkonzeption von BEEP präsentiert und in RFC 3081 [2] hat er die wichtigen Punkte der Abbildung auf TCP-Transport vorgeschlagen. Nach Rose hat BEEP die folgenden Zielsetzungen:

- **Verbindungsorientierung**: Für jede Kommunikation zwischen BEEP-Anwendungen wird eine Verbindung eingerichtet. Die Anwendungen können die Verbindung starten und beenden. Dazwischen können sie Daten austauschen.
- **Nachrichtenorientierung**: Die Anwendungen können durch strukturierte Frames miteinander kommunizieren, d.h. die Anwendungen dürfen die Verbindungs-schnittstelle nicht wissen.
- **Asynchronisation**: BEEP wird nicht auf ein Request-Response-konzept beschränkt, sondern erlaubt die asynchrone Kommunikation zwischen Client und Server.

# 2 Konzeption von BEEP Core

BEEP ist ein Anwendungsprotokoll nach dem Peer-to-Peer-Prinzip. Unter dem Begriff „Peer-to-Peer“ versteht man die bilaterale Kommunikation zwischen zwei oder mehr gleichberechtigten Endpunkten, die beide gleichwertige Fähigkeiten und Verantwortlichkeiten haben. Daher stellt BEEP auch die unmittelbare Verbindung von einem Endpunkt zu einem anderen Endpunkt dar [5]. Daraus folgen die Funktionalitäten von BEEP [4]:

- Trennen einer Nachricht von der Nächsten (Frame),
- Kodierung der Nachrichten,
- Unterstützung mehrfacher logischer Kommunikationskanäle,
- Berichten über Störungen,
- Aushandlung von Verschlüsselungsparametern,
- Aushandlung von Authentisierungsparametern.

## 2.1 Nachrichtaustausch

### 2.1.1 Rolle

Bei Erstellung einer BEEP-Sitzung spielt ein Peer eine Listening-Rolle, wartet also auf eine eingehende Verbindung Im Gegensatz dazu spielt ein anderer Peer, der eine Verbindung zum Listener erstellen will, einen Initiating-Rolle. Ein Peer, welcher einen Datenaustausch starten will, wird als „Client“ bezeichnet. Der andere BEEP-Peer wird entsprechend als „Server“ bezeichnet. Typischweise wird ein Server-Peer eine Listening-Rolle spielen. Aber auf Grund der Eigenschaften von Peer-to-Peer existieren solche Anforderungen bei der BEEP-Verbindung nicht. Im weiteren Verlauf der Arbeit, insbesondere in den Beispielen, werden Listener und Initiator als „L“ und „I“ gekennzeichnet, Client und Server als „C“ und „S“.

### 2.1.2 Austauscharten

BEEP definiert drei Arten des Datenaustauschs, die in Englisch als „Exchange Style“ bezeichnet und nach dem Antworttyp des Servers klassifiziert werden [7]:

- **message/reply**: Der Server führt eine Aufgabe (`MSG`) durch und schickt eine positive Antwort zurück.
- **message/error**: Der Server führt Aufgabe nicht durch und schickt eine negative Antwort (`RPY`) zurück.
- **message/answer**: Der Server schickt null oder mehr Antworten (`ANS`) zurück, die durch eine spezielle Nachricht (`NUL`) abgeschlossen werden. Diese Art kann z.B. für Streaming-Daten angewendet werden.

Die ersten zwei Austauschsarten werden „One-to-One Exchange“ genannt, während die dritte Art als „One-to-Many Exchange“ bezeichnet wird.

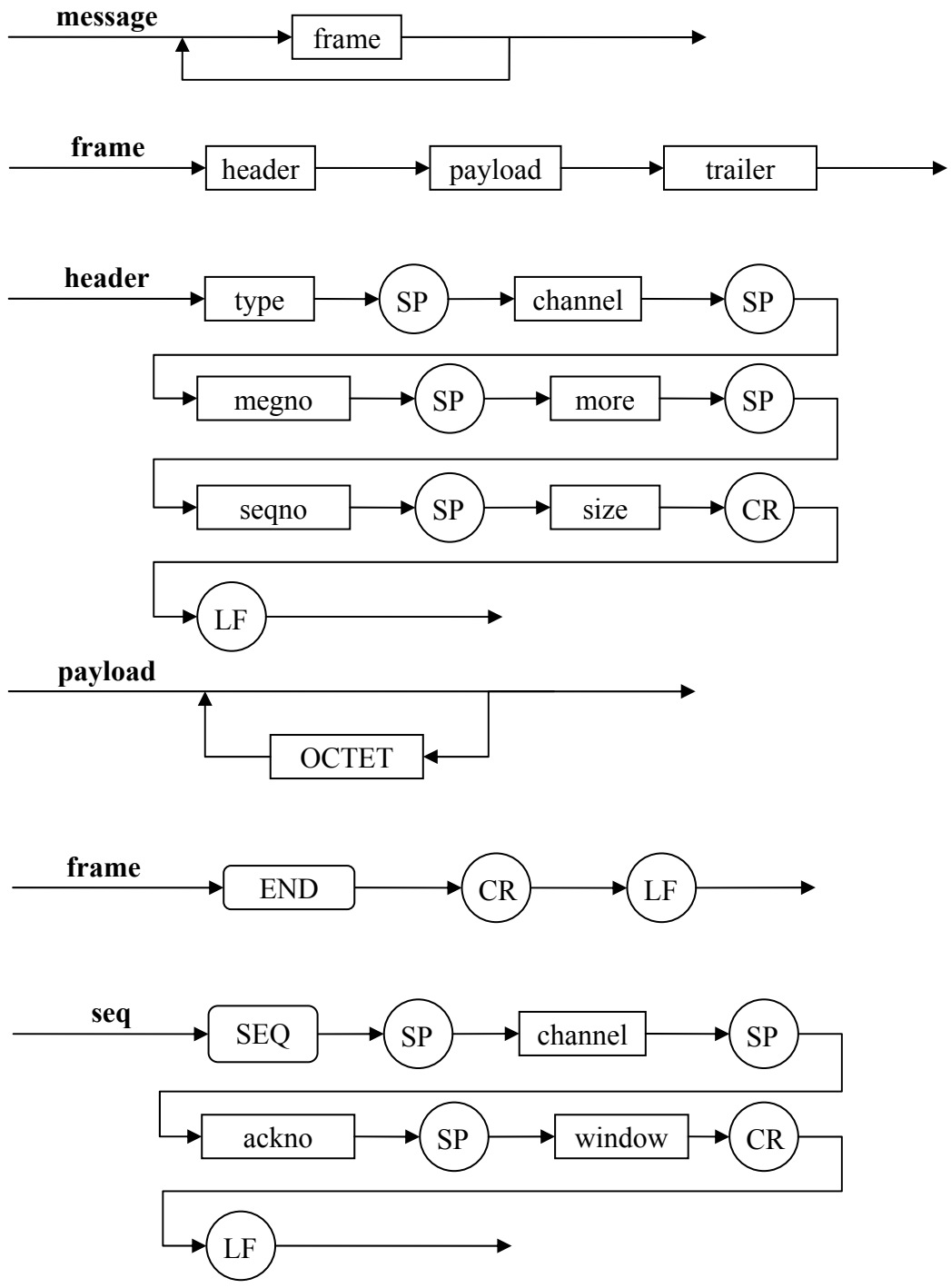


**Abbildung 1 Frame-Syntax**

## 2.2 Frame der Nachrichten

BEEP unterstützt eine große Flexibilität bei der Nachrichtverarbeitung, da die Nachrichten in der Regel durch MIME [8] strukturiert sind. Sie können als Text oder binärer Code repräsentiert werden. Jede Nachricht kann mit „entity-headers“ beginnen, also einem

MIME-Kopf, um den Inhalt der Nachricht zu beschreiben. Der Defaultwert von MIME „Content-Type“ ist „application/octet-stream“ und der von „Content-Transfer-Encoding“ ist „binary“. Die Länge einer Nachricht ist nicht beschränkt und kann beliebig sein. Zur Beschreibung der Nachrichten wird XML benutzt.

Normalerweise wird eine Nachricht in einem Frame geschickt. Aber es ist notwendig, dass eine Nachricht in mehren Frames verteilt wird, z.B. um die zeitliche Verzögerung später zur Verfügung stehender Nachrichtenteile zu vermeiden.

Ein Frame besteht entweder aus BEEP-Daten oder aus Bestandteilen, die durch das Transport-Mapping spezialisiert sind. Die Daten eines Frames bestehen aus drei Teile, nämlich ein Kopf (`header`), seiner Nutzlast (`payload`) und ein Ausschluss (`trailer`). Der Kopf und Ausschluss sind durch die druckbaren ASCII Charakters repräsentiert und enden jeweils mit einer Zeichenketten von <CR> <LF>.

### 2.2.1 Kopf

Der Struktur des Kopfes ist wie folgt definiert (vgl. Abbildung 1):

```
type SP channel SP msgno SP more SP seqno SP size CR LF
```

wobei SP, CR und LF die ACSII Zeichen, bzw. „space“, „carriage-return“ und „linefeed“, kennzeichnen. type spezifiziert die Nachrichtenarten und ist eine 3-Byte Zeichenkette: MSG - „message“, RPY - „reply“, ANS - „answer“, ERR - „error“ oder NUL - „terminator“ [7].

channel identifiziert, auf welchem Kanal die BEEP-Peers miteinander kommunizieren sollen, und wird repräsentiert durch eine nicht-negative Zahl im Intervall $[0...2^{31}-1]$. Hierbei wird der Kanal 0 reserviert für die Kanalverwaltung, der bei dem Erstellen der BEEP-Sitzung implizit geöffnet wird. Demsprechend benötigt die einfachste BEEP-Anwendung nur Kanäle, d.h. Kanal 0 dient der Sitzungsverwaltung. Dabei wird entschieden, ob eine Sitzung erstellt oder geschlossen wird, dementsprechend beschäftigt sich der andere Kanal mit dem Datenaustausch.

msgno identifiziert die Nummern der Nachrichten. Die Nummer wird auch mit einer nicht-negativen Zahl im Intervall $[0...2^{31}-1]$ dargestellt. Die Antworten für dieselbe Nachricht haben die gleiche Nachrichtennummer. Eine Nachrichtennummer, darauf nicht bis zum Empfangen der Nachrichten von ERR, RPY oder NUL-Nachrichten wiederverwendet werden, d.h. die Bearbeitung der Nachricht ist abgeschlossen.

more ist ein Fortsetzungsindikator, der entweder mit „*“ oder mit „.“ gekennzeichnet wird. Der Code „*“ bedeutet, dass dem aktuellen Frame mindestens einem oder mehreren Frames folgen. Der Code „.“ bedeutet, dass dem aktuellen Frame keine Nachfolge hat und der letzte Frame der aktuellen Nachricht ist. Die Dezimalcodes von beiden sind 42 und 46.

seqno identifiziert die Sequenznummer des ersten Oktetts in der Nutzlast und ist eine nicht-negativen Zahl im Intervall $[0...2^{32}-1]$. In den meisten Fällen ist seqno des Frame n die Summe der Längen von Nutzlast vom vorherigen n-1 Frame. Manche Anwendungen können jedoch das BEEP effizient verwenden, um eine große Zahl von überflüssigen Defaultwerte auszulassen, z.B. bei einem verteilten Dateisystemprotokoll kann es vermeidet werden, dass viele Frames mit Nullbyte nacheinander gesendet werden.

`size` kennzeichnet die Anzahl der Oktetts in der Nutzlast eines Frames und muss eine nicht-negative Zahl im Intervall [0...$2^{31}$-1] sein. Die Länge von Kopf und Abschluss des Frames werden nicht mitgezählt.

### 2.2.2 Nutzlast

Die Nutzlast eines Frames liegt zwischen dem Kopf und dem Abschluss, und besteht aus null oder mehren Oktetts. In einer Nutzlast ist jedes Oktett, welches in beiden Richtungen eines Kanals übertragt wird, einer Sequenznummer zugeordnet. Das erste Oktett wird mit der kleinsten Zahl nummeriert. Das erste Oktett in der Nutzlast des ersten Frames hat die Sequenznummer 0 beim Start eines BEEP-Kanals.

### 2.2.3 Abschluss

Der Abschluss eines Frames besteht aus der Zeichenkette `END` <CR> <LF>. Falls beim Empfangen eines Frames kein Anhänger nach der Nutzlast kommt, dann wird die aktuelle Sitzung beendet und der Fehler ggf. in einer Log-Datei eingetragen.

Eine Nachricht kann beispielhaft wie folgt aussehen:

```
C:    MSG 0 1 . 52 120
C:    Content-Type: application/beep+xml
C:    <start number='1'>
C:          <profile uri='http://iana.org/beep/SASL/OTP' />
C:    </start>
C:    END
```

wobei diese Nachricht von einem Client zu einem Server geschickt wird, um einen Kanal zu starten. Der Zielkanal ist Kanal 0 und die Nachrichtnummer ist 1. Diese Nachricht hat einen einzelnen Frame, die Sequenznummer ist 52 und die Anzahl der Oktetts in der Nutzlast des Frames beträgt 120, d.h. vorher sind schon 119 Oktetts bei der Kommunikation übertragen werden.

## 2.3 Sitzungsverwaltung

Alle Kommunikationen einer BEEP-Sitzung können in verschiedenen logischen Kanälen verteilt werden, wie in Abbildung 2 dargestellt. Die beiden Peers der BEEP-Kommunikation benötigen nur eine IP-Verbindung, dann können in dieser Sitzung die neuen Kanäle erzeugt werden [4]. Die Eigenschaft der Kommunikation wird durch die Profile geregelt.

Der erste Kanal, also Kanal 0, hat eine besondere Bedeutung für die BEEP-Sitzung, weil er für die Sitzungsverwaltung zuständig ist. Kanal 0 wird bezeichnet, um Profile zu verhandeln, die Sitzung zu beenden oder fortzusetzen, oder einen anderen Kanal zu eröffnen.

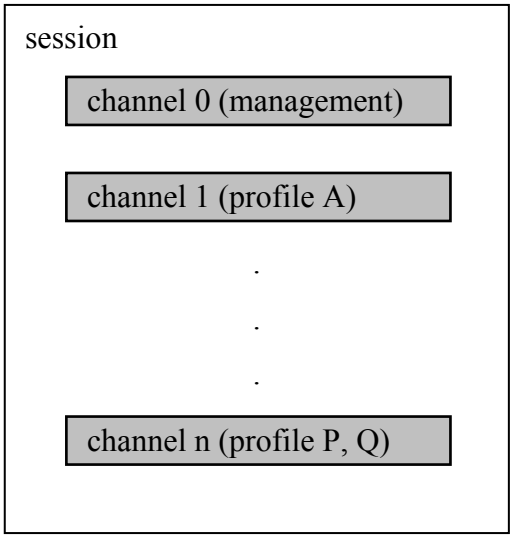


**Abbildung 2: Sitzung, Kanäle und Profile im BEEP**

### 2.3.1 Nachrichten

Nach der Erstellung einer Verbindung werden Nachrichten von verschiedenen Typen zwischen den miteinander kommunizierenden Peers ausgetauscht. Insgesamt gibt es fünf Grundtypen von Nachrichtenelement, die in RFC 3081 [1] vorgegeben und für die BEEP-Sitzung relevant sind, sie sind `greeting`, `start`, `close`, `ok` und `error`. Die Semantik werden im Abschnitt 7.1 von [1] mit DTD ausführlich definiert.

Der Initiator und der Listener der BEEP-Kommunikation können nach verschiedenen Nachrichtenarten die zugelassenen Elemente abschicken. Die Regelung dafür wird in Tabelle 1 dargestellt.

| Rolle | MEG | RPY | ERR |
|---|---|---|---|
| **I und L** | | `greeting` | `error` |
| **I oder L** | `start` | `profile` | `error` |
| **I oder L** | `close` | `ok` | `error` |

**Tabelle 1: Rolle, Nachrichtenarten und Nachrichtenelement**

Eine BEEP-Sitzung ist nach dem Verbindungsaufbau des zugrundeliegenden Transportdienstes erstellt (siehe auch Abschnitt 2.3.2). Dabei wird zuerst eine `greeting`-Nachricht an den Kanal 0 des anderen Peers (Listener) zugeschickt, um ihm die eigene Verfügbarkeit mitzuteilen. Wenn der Listener nicht zur Verfügung steht, schickt er eine negative Antwort zurück, anschließend wird die Sitzung beendet. Wenn der Listener ebenfalls verfügbar ist, dann schickt auch er eine positive Antwort mit `greeting`-Element zurück, so dass die weitere Kommunikation zwischen dem Listener und Initiator durchgeführt werden kann.

Wenn ein BEEP-Peer die aktuelle Sitzung verlassen will, dann schickt er eine `close`-Nachricht an den Kanal 0 vom anderen Peer. Wenn der Server die Beendigung der Sitzung akzeptiert, schickt er eine `ok`-Nachricht zurück und beendet sofort die Verbindung.

Am Anfang der BEEP-Sitzung schickt jeder Peer sofort eine positive Antwort mit einer Begrüßungsnachricht – `greeting` – an den anderen Peer. Die Nachricht enthält ein `greeting`-Element und hat die Nachrichtennummer 0. Durch diese Nachricht wird dem Empfänger mitgeteilt, dass der andere Peer ab jetzt für die Kommunikation bereitsteht, z.B.:

```
L:    RPY 0 0 . 0 110
L:    Content-Type: application/beep+xml
L:
L:    <greeting>
L:          <profile uri='http://iana.org/beep/TLS' />
L:    </greeting>
L:    END
I:    RPY 0 0 . 0 110
I:    Content-Type: application/beep+xml
I:
I:    <greeting />
I:    END
```

In diesem Beispiel werden die zwei Nachrichten von Listener und Initiator nacheinander abgeschickt, aber in der Tat sind die Absendungen voneinander unabhängig.

Falls ein Peer einen neuen Kanal öffnen will, schickt er eine Nachricht mit `start`-Element an den Kanal 0. Um Konflikte der Kanalnummer zu vermeiden, sollte die Kanalnummer vom Initiator eine ungerade positive Integerzahl sein, und die vom Listener eine gerade positive Integerzahl. Eine `start`-Nachricht enthält mindestens einen oder mehr `profile`-Elemente, z.B.:

```
C:    RPY 0 1 . 30 120
C:    Content-Type: application/beep+xml
C:
C:    <start number='1'>
C:          <profile uri='http://iana.org/beep/SASL/OTP' />
C:          <profile uri='http://ibr.cs.tu-bs.de/fangming/contact' />
C:    </start>
C:    END
```

Falls ein Peer einen Kanal schließen will, dann schickt er ein `close`-Element an den Kanal 0. Falls der andere das Schließen des Kanals zulässt, schickt er ein `ok`-Element zurück, z.B.:

```
C:    RPY 0 2 . 390 80
C:    Content-Type: application/beep+xml
C:
C:    <close number='1' code='200' />
C:    END
S:    RPY 0 2 . 210 257
S:    Content-Type: application/beep+xml
S:
S:    <ok />
```

```
S:    END
```

wobei die Attribute `number` die zu schließende Kanalnummer und `code` den Antwortcode darstellt.

Falls ein Peer die Aufgabe des anderen Peer nicht erledigen konnte, z.B. kann er nicht einen neuen Kanal mit Kanalnummer 10 eröffnen, oder gegebenenfalls nicht schließen, dann schickt er eine Fehlermeldung, also eine `error`-Nachricht an den Initiator. Ein Beispiel dafür ist:

```
C:    RPY 0 2 . 390 80
C:    Content-Type: application/beep+xml
C:
C:    <close number='1' code='200' />
C:    END
S:    RPY 0 2 . 211 27
S:    Content-Type: application/beep+xml
S:
S:    <error code='550'>still working</error>
S:    END
```

### 2.3.2 Profile

Ein Profile stellt dar, welche Nachrichten über den Kanal übertragen werden sollen/können. Die Transportsicherheit und die Authentisierung werden durch Profile geregelt. Ein Profile ist identifiziert durch einen URI und regelt die Syntax und Semantik zugelassener Nachrichten [3].

BEEP definiert ein paar Profile für das Kanalmanagement, TLS – „Transport Layer Security" – und Authentisierung, und ermöglicht, dass man die anwendungsorientierten Profile nach praktischen Bedürfnissen definieren kann.

Der TLS Übertragungssicherheit-Profile wird in der BEEP `profile`-Element bei Erstellung eines Kanals als `http://iana.org/beep/TLS` identifiziert. Das entsprechende `profile`-Element im `start`-Elemet kann ein `ready`-Element enthalten. Wenn der Kanal mit Erfolg erstellt ist, dann schickt der BEEP-Peer ein `proceed`-Element in dem entsprechenden `profile`-Element, z.B.:

```
C:    RPY 0 1 . 52 80
C:    Content-Type: application/beep+xml
C:
C:    <start number='1'>
C:          <profile uri='http://iana.org/beep/TLS'>
C:                <![CDATA[<ready />]]>
C:          </profile>
C:    </start>
C:    END
S:    RPY 0 2 . 21 273
S:    Content-Type: application/beep+xml
S:
S:    <profile uri='http://iana.org/beep/TLS'>
S:          <![CDATA[<proceed />]]>
```

```
S:     </profile>
S:     END
```

Das `ready`-Element hat ein optionales Attribut `version` und besitzt kein Inhalt. Das version-Attribut definiert die älteste Version vom TLS, die geeignet für die aktuelle Nutzung ist. Das `proceed`-Element hat kein Attribut und auch keine Inhalt. Es ist eine Antwort zum `ready`-Element.

## 2.4 Abbildung auf TCP

Wie in Abschnitt 2.3 schon erwähnt, kann BEEP auf einen zugrundeliegenden und verbindungsorientierten Transportdienst abgebildet werden, z.B. auf TCP (siehe Abbildung 3). Die Konzeption der Abbildung – „mapping“ – von BEEP auf TCP ist in RFC 3081 [2] verdeutlicht. Die Abbildung auf TCP beinhaltet zwei Bereiche: Sitzungsverwaltung und atenaustausch.

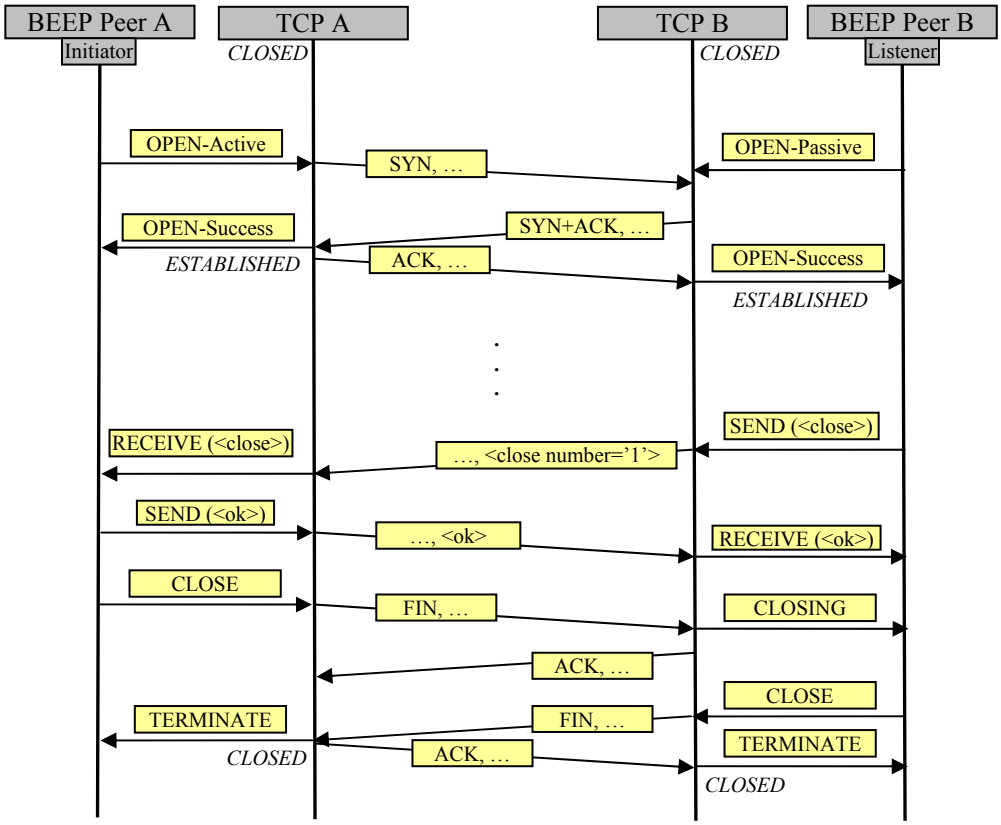


**Abbildung 3: Sitzungsverwaltung bei Abbildung auf TCP**

### 2.4.1 Sitzungsverwaltung

Die Sitzungsverwaltung wird direkt auf eine TCP-Verbindung abgebildet. Der Peer, welcher ein passives TCP OPEN ausgeführt, wird als Listener bezeichnet. Der Peer, welcher ein aktives TCP OPEN sendet, wird als Initiator bezeichnet.

Der Ablauf der Sitzungsverwaltung wird in Abbildung 3 dargestellt. Dabei sind Sendungen der `greeting`-Nachricht von Initiator und Listener nicht aufeinanderfolgend, sondern voneinander unabhängig.

### 2.4.2 Nachrichtenaustausch

Im Gegensatz zur Sitzungsverwaltung, ist die Abbildung des Nachrichtenaustauschs etwas aufwändiger. Zum zuverlässigen Empfangen und Senden der Nachrichten werden die SEND- und RECEIVE-Funktionen von TCP (siehe Abbildung 4) genutzt.

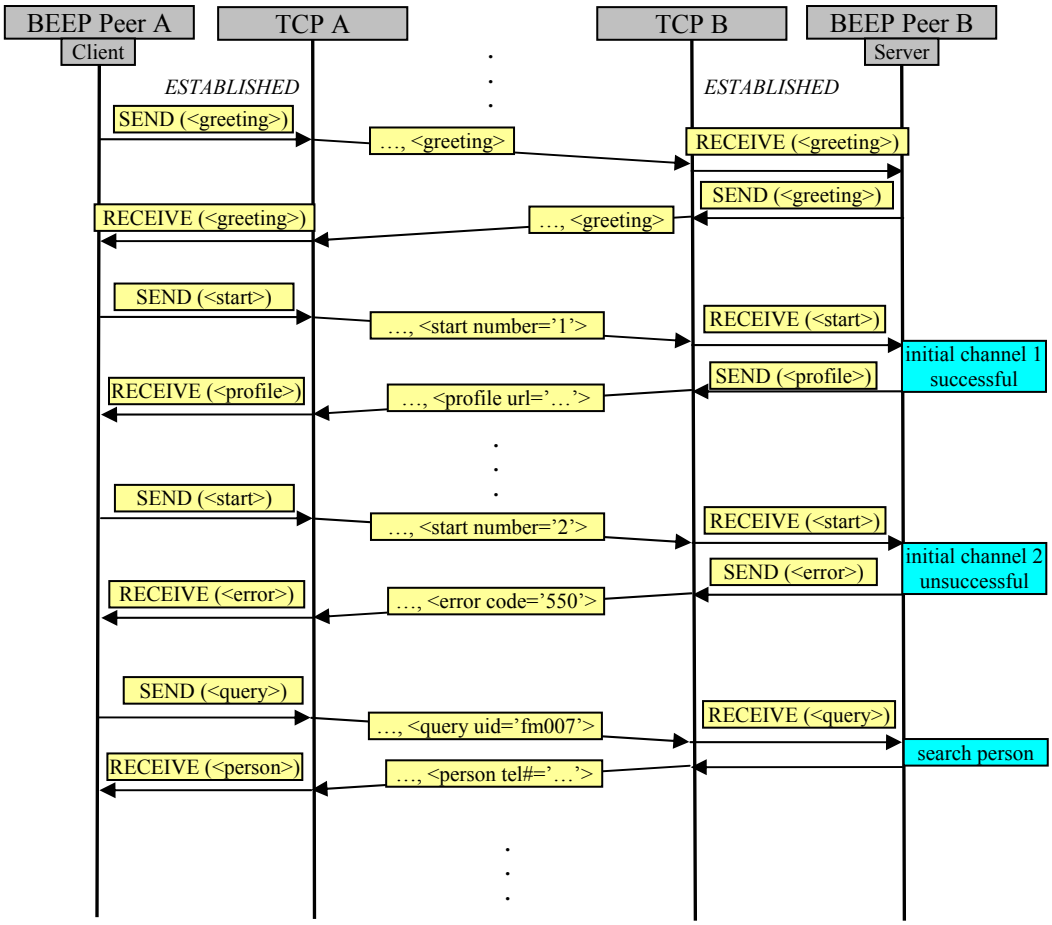


**Abbildung 4: Nachrichtenaustausch bei Abbildung auf TCP**

Eine BEEP-Sitzung kann mehrere Kanäle verwenden. Daher hat BEEP einen Mechanismus, um die Störungen auf jedem Kanal voneinander unabhängig behandeln zu können: Die auf „Fenstern" basierte Flusskontrolle. Jeder Kanal hat ein verschiebbares Fenster, welches die zulässige Anzahl der Oktetts bei Nachrichtenaustausch festlegt. Die Größe des Fensters kann sich durch Senden eines `SEQ`-Frames dynamisch ändern.

Ein `SEQ`-Frame wird in Abbildung 1 definiert. Sie hat drei Parameter: `channel`, `ackno` und `window`. `ackno` und `window` sind identisch wie `seqno` und `size`, die in Abschnitt 2.2.1 definiert worden sind.

# 3 Implementierung der Konzeption mit Java BEEP

Zahlreiche Bibliotheken von BEEP werden zur Verfügung gestellt. Beepcore.org [9] ist die offizielle Homepage von BEEP und dort sind ein paar Implementierungen von BEEP aufgeführt. Bislang gibt es schon die Implementierungen für Java, C und TCL, weitere Implementierungen für Python und Ruby werden entwickelt. Hier wird Java BEEP anhand eines Beispiels betrachtet.

Zunächst wollen wir darauf fokussieren, wie die einzelnen Bestandteile der BEEP-Konzeption in Java BEEP implementiert sind:

- `Session`: Jede BEEP-Verbindung benötigt zuerst eine Instanz dieser Klasse. Ein Objekt der Sitzung hat die Funktionen, die Profile zu akzeptieren und die Kanäle zu starten.
- `Channel`: Ein Objekt von Kanal wird zurückgeliefert, wenn die Methode `startChannel()` von einer Sitzung aufgerufen ist. Es kann `MSG`-Nachricht schicken durch die Methode `sendMSG()`.
- `Profile`: Der Begriff Profile ist als ein Interface definiert und hat nur eine Methode, die für das spezifischen Profile einen Listener des Ereignisses Startkanal zurückgibt. Alle Implementierungen von `Profile` müssen das Interface implementieren. Ein Startkanal-Listener akzeptiert das Objekt von Kanal und kann für jede Nachricht oder jeden Frame einen Listener registrieren.
- `Message`: Diese Klasse kapselt alle Nachrichtenarten, nämlich `MSG`, `RPY`, `ANS`, `ERR` und `NUL`. Es kann über seinen Inhalt abgefragt werden, und hat die Methoden: `sendMSG()`, `sendRPY()`, `sendANS()`, `sendERR()` und `sendNUL()`.
- `Reply`: Ein Objekt von `Reply` kann die asynchrone Nachricht vom anderen Peer behandeln. Seine repräsentative Methode ist `getNextReply()`, die eine weitere Nachricht zurückgeben kann.

Der Nutzen der oben genannten Klasse ist abhängig von der Rolle des Peers, welcher in die BEEP-Interaktion eingetreten ist. Ein Server instanziert den Listener für `MSG` für einen Kanal. Ein Client initialisiert den `ReplyListener` bei Abschicken der `MSG`. Hierbei wird anhand eines Beispiels aus [6] der Kommunikationsmechanismus erläutert.

## 3.1 Client

Das clientseitigen Programm wird in Methode `main()` implementiert. Zuerst wird eine Verbindung mit dem Server `dragon`, der auf Port `8888` läuft, erstellt:

```
import org.beepcore.beep.core.*;
import org.beepcore.lib.Reply;
import org.beepcore.transport.tcp.*;
...
public static void main (String args[]) {
      Session session;
      try {
```

```
        session = AutomatedTCPSessionCreator.initiate("dragon", 8888,
                                                new ProfileRegistry());
    } catch (BEEPException e) {
        // exception handling
    }
```

Durch Nutzen einer Hilfsklasse des Java API - `AutomatedTCPSessionCreator` - wird eine `Session` initialisiert. Dann wird es versuchen, einen Kanal zu starten:

```
Channel channel;
try {
        channel = session.startChannel(
        "http://ibr.cs.tu-bs.de/fangming/ADDRESSBOOK"); //profile uri
} catch (BEEPError e) {
        // catch and deal with errors
        System.out.print("error code: "+ e.getCode());
} catch (BEEPException e) {
        // catch lower level exceptions
}
```

Wenn es klappt, dann entsteht ein Objekt von `Channel`, welches ein die Kommunikation unterstützende `Profile` benutzt. Nun wird eine Nachricht abgeschickt:

```
String message =  "<person>"+
                  "     <name>Fangming</name>" +
                  "     <tel>0049-531-2145611</tel>"+
                  "</person>";
ByteDataStream  msgst  =  new  ByteDataStream(request.getByes("utf-
16"));
msgst.setContentType("text/xml");
msgst.setTransferEncoding("utf-16");
Reply reply = new Reply();
channel.sendMSG(msgst, reply);
```

Die Nachricht wird hier mit `utf-16` kodiert, und zwar mit der MIME-Type `text/xml` versehen. Am Ende dieses Blocks ist die Nachricht durch `sendMSG()` abgeschickt worden. Dann wird die Antwort von dem Server erwartet:

```
while (true) {
        Message repmsg = reply.getNextReply();
        DataStream ds = repmsg.getDataStream();
        InputStream is = ds.getInputStream();
        // the reply is a stream, not a string
        byte inputbuf[]=new byte[256];
        int inputlen;
        String reptxt;
        while (ds.isComplete() == false || is.available() > 0) {
              inputlen = is.read(inputbuf);
              if (inputlen > 0) {
                    reptxt = reptxt.concat(new String(inputbuf, 0,
                    inputlen, "utf-16");
              }
        }
```

```
            System.out.println("got back:" +reptxt);
            Thread.currentThread.sleep(6000);
      } // end while
} // end main
```

Hierbei gilt es die wichtige Aufmerksamkeit, dass es unnötig ist, die Eigenschaft einer Nachricht zu bestimmen.

## 3.2 Server

Auf der Seite des Servers wird eine Klasse `AddressbookProfile` definiert, welche die Interface `Profile`, `StartChannelListener` und `MessageListener` implementiert.

```
public class AddressbookProfile implements
      Profile, StartChannelListener, MessageListener {
      ...
      public StartChannelListener init(String uri,
            ProfileConfiguration config) throws BEEPException {
            return this;
      }
      public void startChannel(Channel channel, String encoding,
            String data) throws StartChannelException {
            channel.setDataListener(this);
      }
      public void closeChannel(Channel channel)
            throws CloseChannelException {
            channel.setDataListener(null);
      }
```

Dabei sind die Methoden `init()`, `startChannel()` und `closeChannel()` definiert. Weil für jede Sitzung nur ein `MessageListener` initialisiert wurde, wird das Objekt von allen Kanälen benutzt.

```
      public void receiveMSG(Message message)
            throws BEEPError, AbortChannelException {
            DataStream ds = message.getDataStream();
            AddressbookEntry newentry;
            String replytxt = "<person uid=\"" +
                              String.valueOf(newentry.getUID()) + \""/>";
            DataStream reply=new ByteDataStream(
                  replytxt.getBytes("utf-16"));
            reply.setContentType("text/xml");
            try {
                  message.sendRPY(reply);
            } catch (BEEPException e) {
                  // exception handling
            }
      }
      ...
} // end class AddressbookProfile
```

In der Methode `receiveMSG()` wird zuerst die Nachricht von Client empfangen, dann suchen das Ergebnis. Wenn es gefunden ist, versucht der Server eine positive Antwort an den Client zurückzuschicken.

# 4 Zusammenfassung

BEEP ist ein verbindungsorientiertes Protokoll und unterstützt die asynchrone Interaktion. Die zu übertragenden Nachrichten werden in Frames verteilt, und mit verschiedenen Austauscharten zwischen BEEP-Peer ausgetauscht. Die Daten werden mit XML kodiert, dann bietet BEEP flexibleres Datenaustausch an. Dementsprechend wurden und werden weiteren Profiles, z.B. für SOAP entwickelt.

BEEP hat die Fähigkeit von Peer-to-Peer, Client-to-Server oder Server-to-Server und ermöglicht, dass in die einzelne Sitzung mehrfache Kanäle eröffnet werden. Durch die Standard-Profile werden die Transportsicherheit und Authentisierung der Sitzung unterstützt. Der „Missbrauch“ von http für Anwendungsprotokolle kann durch BEEP elegant abgelöst werden.

# Literatur


[1] M. Rose, The Blocks Extensible Exchange Protocl Core, RFC 3080, März 2001

[2] M. Rose, Mapping the BEEP Core onto TCP, RFC 3081, März 2001

[3] IETF BEEP (RFC 3080) – A Framework for Next Generation Application Protocols, technical Whitepaper, Mai 2002, URL: http://www.clipcode.com

[4] E. Dumbill, Bird's-eye BEEP, Part 1 of an introduction to the Blocks Extensible Exchange Protocol standard of the IETF, Dezember 2001, URL: http://www-900.ibm.com/developerWorks/cn/xml/x-watch/part1/index_eng.shtml

[5] T. Wieland und E. Chtchibina, Peer Preview – Plattformen für Peer-to-Peer Netzte.

[6] E. Dumbill, Worm's-eye BEEP, Part 2 of an introduction to the Blocks Extensible Exchange Protocol standard of the IETF, März 2001, URL: http://www-900.ibm.com/developerWorks/cn/xml/x-watch/part2/index_eng.shtml

[7] R. Salz, Beep BEEP!, Oktober 2002, URL: http://www.xml.com/pub/a/2002/10/16/ ends.html

[8] Freed, N. und Borendstein, N., Multipurpose Internet Mail Extensions, RFC 2046, November 1996

[9] Home of BEEP, URL: http://www.beepcore.org/beepcore/home.jsp

[10] E. O' Tuathail und M. Rose, Using SOAP in BEEP, RFC 3288, Juni 2002