Eine Socket-Abstraktion in Z

Jan Peleska
jp@informatik.uni-bremen.de

Wir geben hier die wesentlichen Funktionen der Stream Sockets in Form abstrakter Z-Spezifikationen wieder. Dabei ist zu beachten, daß die Z-Schemata nur partiell – nämlich für den Fall der fehlerfreien Funktionsausführung – definiert sind.

1 Basistypen und Konstanten

Folgende Basistypen und Konstanten werden in den Z-Spezifikationen weiter unten verwendet:

Menge der Datenbytes:

$$[BYTE]$$

Menge aller möglichen Prozeßidentifikationen:

$$[PID]$$

Menge aller möglichen Dateideskriptoren:

$$[FD]$$

Menge aller möglichen eindeutigen Dateiidentifikationen (z. B. Host-Id plus Inode-Nummer plus Device-Id):

$$[FILEID]$$

Menge aller möglichen Socketadressen (z. B. IP-Nummer plus Port):

$$[SOCKETAD]$$

Maximale Anzahl Dateideskriptoren, die einem Prozeß zugeordnet werden können:

$$\mid \; maxdes : \mathbb{N}$$

Länge einer Connect-Message:

$$lconmsg : \mathbb{N}$$

Hilfsfunktion *toConmsg* wandelt Adresse in zugehörige Byte-Sequenz der Connect-Message. Hilfsfunktion *firstAd* gibt aus einer Byte-Sequenz die erste Socket-Adresse zurück, wenn die ersten Bytes eine solche definieren (da nicht jede Byte-Sequenz eine Adresse definiert, ist dies nur eine partielle Definition). Die Hilfsfunktion sind zueinander invers.

$$\begin{array}{|l}
toConmsg : SOCKETAD \rightarrow \mathrm{seq}\ BYTE \\
firstAd : \mathrm{seq}\ BYTE \nrightarrow SOCKETAD \\
\hline
\mathrm{ran}\ toConmsg = \mathrm{dom}\ firstAd \\
\forall a : SOCKETAD \bullet firstAd(toConmsg(a)) = a
\end{array}$$

Erste Connect-Message wird aus Connect Queue entfernt; die um diese Message verkürzte Queue wird zurückgegeben:

$$removeFirstAd : \mathrm{seq}\ BYTE \rightarrow \mathrm{seq}\ BYTE$$

Maximalgröße des Backlogparameters im Kernel:

$$maxbl : \mathbb{N}$$

2 Socket-Zustandsraum

Abstraktion des Zustandsraums für die Kommunikation mit Sockets:

$$\begin{array}{|l|}
\hline
SocketStateSpace \\
\hline
fd : PID \rightarrow \mathbb{P}(FD) \\
fp : PID \times FD \nrightarrow FILEID \\
f : FILEID \nrightarrow \mathrm{seq}\ BYTE \\
sa : SOCKETAD \nrightarrow PID \times FD \\
bl : PID \times FD \nrightarrow \mathbb{N} \\
ep : SOCKETAD \nrightarrow SOCKETAD \\
\hline
\mathrm{dom}\ bl \subseteq \{p : PID;\ d : FD \mid d \in fd(p) \bullet (p, d)\} \\
(\forall p : PID;\ d : FD \bullet d \in fd(p) \\
\qquad \Rightarrow (p, d) \in \mathrm{dom}\ fp \wedge fp(p, d) \in \mathrm{dom}\ f) \\
(\forall p : PID;\ d : FD \bullet (p, d) \in \mathrm{dom}\ bl \\
\qquad \Rightarrow \#(f(fp(p, d)))\ \mathsf{div}\ lconmsg \leq bl(p, d)) \\
(\forall a_1, a_2 : SOCKETAD \bullet ep(a_1) = a_2 \Leftrightarrow ep(a_2) = a_1) \\
\hline
\end{array}$$

Dabei haben die Komponenten des Zustandsraums folgende Bedeutung:

- fd: Die einem Prozeß zugeordneten Dateideskriptoren. $fd(p) = \emptyset$, falls Prozeß p keine Datei geöffnet hat.
- fp: Filepointer, d. h. Zuordnung der Paare (Prozeßidentifikation, Dateideskriptor) zu einer systemweit eindeutigen Dateiidentifikation.
- f: Zuordnung der Dateiidentifikationen zu ihrem Inhalt. Der Inhalt wird als Bytestream, d. h. Folge von Bytes interpretiert.
- sa: Zuordnung definierter Socketadressen zu den zugehörigen Paaren (Prozeßidentifikation,Dateideskriptor).
- bl: Backlog, d. h. Anzahl der für den mit (Prozeßidentifikation,Dateideskriptor) bezeichneten Socket zu puffernden Connect-Requests. Das Backlog ist nur für sockets definiert, die durch einen `listen`-Befehl als Connect-Queues für den Verbindungsaufbau beim Server eingerichtet werden.
- ep: Kennzeichnung der *End-Point Connections*, d. h., der Paare von Adressen, zwischen denen eine Socket-Verbindung mit `connect()/accept()` aufgebaut wurde.

3 Systeminitialisierung

Beim Start des Systems gibt es noch keine offenen Dateien etc:

$$
\begin{array}{|l}
\textit{Init} \\
\hline
\textit{SocketStateSpace}' \\
\hline
(\forall p : \textit{PID} \bullet \textit{fd}'(p) = \emptyset) \\
\mathrm{dom}\ \textit{fp}' = \emptyset \\
\mathrm{dom}\ f' = \emptyset \\
\mathrm{dom}\ \textit{sa}' = \emptyset \\
\mathrm{dom}\ \textit{bl}' = \emptyset \\
\mathrm{dom}\ \textit{ep}' = \emptyset \\
\hline
\end{array}
$$

4 Socket-Operationen

4.1 `socket()`

Die Operation `socket()` liefert einen Dateideskriptor, der den Socket für den Prozeß eindeutig identifiziert. Zu beachten ist dabei, daß Dateideskriptoren zwar prozeßspezifisch sind, aber mehrere Prozesse im System gleiche Dateideskriptoren verwenden können.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Socket}} \\
\hline
\Delta \mathit{SocketStateSpace} \\
p? : \mathit{PID} \\
d! : \mathit{FD} \\
\hline
\#(\mathit{fd}(p?)) < \mathit{maxdes} \\
d! \notin \mathit{fd}(p?) \\
\mathit{fd}' = \mathit{fd} \oplus \{p? \mapsto \mathit{fd}(p?) \cup \{d!\}\} \\
(\mathbf{let}\ z == (\mu\, x : \mathit{FILEID} \mid x \notin \mathrm{dom}\ f) \bullet \\
\qquad\qquad\qquad \mathit{fp}' = \mathit{fp} \oplus \{(p?, d!) \mapsto z\} \wedge f' = f \oplus \{z \mapsto \langle\ \rangle\}) \\
\mathit{sa}' = \mathit{sa} \\
\mathit{bl}' = \mathit{bl} \\
\mathit{ep}' = \mathit{ep} \\
\hline
\end{array}
$$

Vorbedingung der Operation ist die Verfügbarkeit weiterer neuer Dateideskriptoren für den Prozeß $p?$. *Socket* liefert dann einen neuen Dateideskriptor zurück und verändert die Abbildung fd so, daß das Bild des Prozeßidentifikators $p?$ unter fd um den neuen Deskriptor ergänzt wird. Eine weitere Wirkung der Operation *Socket* ist die Veränderung der Komponenten fp und f: die Filepointer werden ergänzt um eine Komponente für den neuen Dateideskriptor (relativ zum Prozeß $p?$), der auf einen neuen – d.h. noch nicht in dom f vorkommenden – Dateiidentifikator abgebildet wird; diesem Dateiidentifikator wird in f ein leerer Byte-Strom zugeordnet.

4.2 `bind()`

Die Operation *Bind* bindet einen neuen Namen, d.h. eine Socketadresse an einen unbenannten Socket.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Bind}} \\
\hline
\Delta SocketStateSpace \\
p? : PID \\
d? : FD \\
a? : SOCKETAD \\
\hline
d? \in fd(p?) \\
a? \notin \operatorname{dom} sa \\
\\
fd' = fd \\
fp' = fp \\
f' = f \\
sa' = sa \oplus \{a? \mapsto (p?, d?)\} \\
bl' = bl \\
ep' = ep \\
\hline
\end{array}
$$

Vorbedingung für die Anwendung von *Bind* ist daß

- der Dateideskriptor ein zulässiger für den angegebenen Prozeß ist, und
- die Socketadresse eine neue ist[1].

Die einzige veränderte Komponente ist die Abbildung sa, wo ein neuer Eintrag das Paar $(p?, d?)$ an die neue Adresse bindet, also den Deskriptor über die Adresse zugänglich macht.

4.3 `listen()`

Diese Operation schafft die Voraussetzungen für die eigentliche Kommunikation auf der Seite des Servers: die Connect-Queue wird für den Socket eingerichtet, wobei auch die Maximalzahl der Backlog-Einträge festgelegt wird[2].

[1] Wir abstrahieren hier von der internen Struktur der Socketadressen: Bei IP-Adressen z. B. besteht diese aus einem Paar (IP-Adresse,Port), wobei die IP-Adresse immer die des lokalen Rechners ist. Diese Bedingung ist in unserer Abstraktion *SOCKETAD* nicht modelliert.

[2] Bei Solaris 2.5 ist derzeit *keine* Obergrenze für die Größe des Backlog-Parameters im Funktionsaufruf definiert. Wir geben hier die übliche Spezifikation, die $b?$ nur bis zu einem Maximalwert einstellt und größere Werte ignoriert. Bei Linux ist zur Zeit eine Maximalgröße des Backlogs von `SO_MAXCONN = 128` Einträgen angegeben. Die Manualpages geben keine Auskunft darüber, wie die Wirkung von $b? \leq 0$ ist. Wir nehmen an: Fehler.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Listen}} \\
\hline
\Delta \mathit{SocketStateSpace} \\
p? : \mathit{PID} \\
d? : \mathit{FD} \\
b? : \mathbb{N} \\
\hline
b? > 0 \\
d? \in \mathit{fd}(p?) \\
\\
\mathit{fd}' = \mathit{fd} \\
\mathit{fp}' = \mathit{fp} \\
f' = f \\
\mathit{sa}' = \mathit{sa} \\
\mathit{bl}' = \mathit{bl} \oplus \{(p?, d?) \mapsto (\textbf{if } b? < \mathit{maxbl} \textbf{ then } b? \textbf{ else } \mathit{maxbl})\} \\
\mathit{ep}' = \mathit{ep} \\
\hline
\end{array}
$$

Voraussetzung für die Anwendung der Operation *Listen* ist, daß

- das Argument $b?$ zur Festlegung der maximalen Anzahl möglicher Backlog-Einträge eine natürliche Zahl ist, und
- der Dateideskriptor ein zulässiger für den Prozess ist.

Listen wirkt auf die bl-Komponente des *SocketStateSpace*. Sie wird durch den Eintrag für das Paar $(p?, d?)$ ergänzt, wobei die Backlog-Länge sich als Minimum von $b?$ und der globalen Konstanten $maxbl$ ergibt.

4.4 `connect()` und `accept()`

Socketfunktion `connect()` wird vom Client verwendet, um die Verbindung mit dem Server herzustellen. Wir modellieren sie durch zwei Schemata, welche die beiden Phasen des Systemaufrufs darstellen:

- In Phase 1 wird der Verbindungsaufbauwunsch des Clients in die Connect-Queue des Servers eingetragen. Der Client ist jetzt im Systemaufruf blockiert, der Server hat noch nicht das zugehörige `accept()` durchgeführt. Dies wird durch das Z-Schema *Connect*0 modelliert.
- In Phase 2 wird – ausgelöst durch den entsprechenden `accept()`-Systemaufruf beim Server – der Verbindungsaufbauwunsch aus der Connect-Queue des Servers entfernt. Mit dem synchronen Terminieren von `connect()`

und `accept()` ist eine Punkt-zu-Punkt Verbindung zwischen dem Client-Socket und dem neuen, als Rückgabewert des `accept()`-Aufrufs erhaltenen Socket beim Server hergestellt. Schema *ConTerm* beschreibt das Terminieren des `connect()` beim Client. Die Gleichzeitigkeit des Terminierens mit dem `accept()`-Aufruf wird unten durch die Z-Schemakonjunktion *AcceptConTerm* spezifiziert.

$$
\begin{array}{|l}
\hline
\textit{Connect0} \\
\Delta \textit{SocketStateSpace} \\
p? : \textit{PID} \\
d? : \textit{FD} \\
a? : \textit{SOCKETAD} \\
\hline
d? \in \textit{fd}(p?) \\
a? \in \operatorname{dom} sa \\
sa(a?) \in \operatorname{dom} bl \\
\#(f(\textit{fp}(sa(a?)))) \mathbin{\mathsf{div}} \textit{lconmsg} < bl(sa(a?)) \\
\\
\textit{fd}' = \textit{fd} \\
\textit{fp}' = \textit{fp} \\
(\mathbf{let}\ \textit{fid} == \textit{fp}(sa(a?));\ a0 == (\mu\, a : \textit{SOCKETAD} \mid a \notin \operatorname{dom} sa) \bullet \\
\qquad f' = f \oplus \{\textit{fid} \mapsto f(\textit{fid}) \frown \textit{toConmsg}(a0)\} \land \\
\qquad sa' = sa \oplus \{a0 \mapsto (p?, d?)\}) \\
bl' = bl \\
ep' = ep \\
\hline
\end{array}
$$

Die Voraussetzungen für das *Connect*0 sind:

- Der Dateideskriptor $d?$ ist zulässig für den Prozeß.
- $a?$ ist eine eingetragene Socketadresse, über welcher ein Server Verbindungsaufbauwünsche bearbeitet, also einen `listen()`-Aufruf gemacht hat. Letzteres ist an $sa(a?) \in \operatorname{dom} bl$ erkennbar.
- Die zur Socketadresse gehörige Connect-Queue enthält weniger als die erlaubte Anzahl von Messages.

Operation *Connect*0 ordnet dem Client-Socket eine neue Socketadresse $a0$ zu. Die Ausführung von *Connect*0 verändert weiterhin den Inhalt der zum Server-Socket gehörenden Dateien durch Anhängen des Byte-Stroms, der sich aus Konvertierung der Message ergibt, die in diesem Fall aus der neu zugeordneten Socketadresse $a0$ besteht. Die übrigen Komponenten bleiben unverändert.

Der beim Server erfolgende Systemaufruf `accept()` stellt die Verbindung endgültig her, sobald der durch den oben beschriebenen Connect-Befehl erzeugte Eintrag der erste in der Connect-Queue des Servers ist. Dazu wird ein neuer Socket kreiert und über diesen die Verbindung zwischen Client und Server hergestellt. Die Operation wird durch zwei Schemata beschrieben, wobei das erste (*Accept*0) die Vorbedingungen für den erfolgreichen Systemaufruf spezifiziert und – dies wird unten im Schema *AcceptConTerm* spezifiziert – in Konjunktion mit Schema *Socket* der neue Socket für den Server erzeugt wird.

$$
\begin{array}{|l}
\textit{Accept}0 \\
\hline
\textit{SocketStateSpace} \\
p? : \textit{PID} \\
d? : \textit{FD} \\
d' : \textit{FD} \\
d! : \textit{FD} \\
\hline
d? \in \textit{fd}(p?) \\
(p?, d?) \in \operatorname{dom}\ bl \\
\#(f(\textit{fp}(p?, d?))) \mathbin{\mathsf{div}} \textit{lconmsg} > 0 \\
d' = d! \\
\hline
\end{array}
$$

Schema *Accept*0 fordert, daß $d?$ als Dateideskriptor für $p?$ definiert ist und für $(p?, d?)$ eine Connect-Queue definiert wurde (d. h., es gilt $(p?, d?) \in \operatorname{dom}\ bl$). Der in Konjunktion mit Schema *Socket* erzeugte neue Dateideskriptor $d!$ wird an die neue Schemakomponente d' gebunden und dann vom Schema *Accept*1 weiter benutzt, wie unten aus Schemagleichung *AcceptConTerm* ersichtlich.

$$
\begin{array}{|l}
\textit{Accept}1 \\
\hline
\Delta \textit{SocketStateSpace} \\
p? : \textit{PID} \\
d? : \textit{FD} \\
\textit{aclient}! : \textit{SOCKETAD} \\
d : \textit{FD} \\
\hline
(\mathbf{let}\ \textit{fid} == \textit{fp}(p?, d?);\ a0 == (\mu\, a : \textit{SOCKETAD} \mid a \notin \operatorname{dom}\ sa) \bullet \\
\qquad \textit{aclient}! = \textit{firstAd}(f(\textit{fid})) \land \\
\qquad f' = f \oplus \{\textit{fid} \mapsto \textit{removeFirstAd}(f(\textit{fid}))\} \land \\
\qquad ep' = ep \oplus \{a0 \mapsto \textit{aclient}!, \textit{aclient}! \mapsto a0\} \land \\
\qquad sa' = sa \oplus \{a0 \mapsto (p?, d)\}) \\
\hline
\end{array}
$$

Schema *Accept*1 spezifiziert die Wirkung des Accept-Befehls beim Server: Die Socketadresse *aclient*! des Clients wird ausgegeben und aus der Connect-Queue des Servers gelöscht. Der neue von *Socket* erzeugte Dateidescriptor

d wird dem Server zugeordnet und dafür auch eine neue Socketadresse $a0$ erzeugt. In Funktion ep wird die neue Zuordnung der Kommunikationsendpunkte $aclient!$ und $a0$ vermerkt.

Das erfolgreiche Terminieren des Connect-Befehls beim Client wird durch Schema $ConTerm$ modelliert:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\; ConTerm \;}} \\
SocketStateSpace' \\
pclient? : PID \\
dclient? : FD \\
aclient! : SOCKETAD \\
returnCode : \{-1, 0\} \\
\hline
returnCode = 0 \\
\hline
\end{array}
$$

Der synchrone Ablauf von $Accept$ und dem Terminieren des zugehörigen $Connect$-befehls wird durch Schemagleichung $AcceptConTerm$ spezifiziert: Wenn die in $Accept0$ definierten Bedingungen erfüllt sind und mit $Socket$ eine neuer Socket für den Server erzeugt werden konnte, verbindet $Accept1$ die Kommunikationsendpunkte, und Client und Server terminieren synchron.

$$AcceptConTerm \mathrel{\widehat{=}} (Accept0 \land Socket) \fatsemi (Accept1 \land ConTerm)$$

4.5 `write()`

Die Operation `write()` schreibt die Daten aus einem Puffer in die durch den Socket definierte Datei beim Kommunikationspartner. Die Schreiboperation setzt voraus, daß die Verbindung zwischen den beiden Kommunikationsendpunkten bereits aufgebaut wurde: Man kann kein `write()` auf einem existierenden lokalen Socket durchführen, wenn hierzu noch kein Kommunikationspartner mittels `connect()/accept()` identifiziert wurde.

Das folgenden Schema beschreibt nur den Normalfall einer Anwendung von `write()`.

$$
\begin{array}{|l}
\hline
\textit{Write} \\
\Delta SocketStateSpace \\
p? : PID \\
d? : FD \\
lbuffer? : \mathbb{N} \\
buffer? : \mathrm{seq}\, BYTE \\
bytesWritten! : \mathbb{N} \\
\hline
d? \in fd(p?) \\
(\exists\, a : \mathrm{dom}\ ep \bullet sa(a) = (p?, d?)) \\
(\mathbf{let}\ fid == fp(sa(ep(\mu\, a : SOCKETAD \mid sa(a) = (p?, d?)))) \bullet \\
\qquad f' = f \oplus \{fid \mapsto f(fid) \frown ((1 \mathrel{..} lbuffer?) \lhd buffer?)\}) \\
bytesWritten! = lbuffer? \\
fd' = fd \\
fp' = fp \\
sa' = sa \\
bl' = bl \\
\hline
\end{array}
$$

Für die Anwendbarkeit von *Write* muß $d?$ ein für $p?$ definierter Dateideskriptor sein. Der Kommunikationspartner muß mittels Abbildung *ep* identifizierbar sein. Sind diese Voraussetzungen erfüllt, ist die Wirkung von Operation *Write* wie folgt:

- Der Rückgabewert $bytesWritten!$ enthält die Länge des gesendeten Puffers – dies ist im Normalfall immer die übergebene Länge $lbuffer?$.
- Bei der durch *fid* identifizierten Datei des via *ep* ermittelten Kommunikationspartners werden die in $buffer?$ stehenden Daten angehängt.

4.6 `read()`

Die Operation `read()` liest Daten aus der dem lokalen Socket zugeordneten Datei in einen Puffer. Voraussetzungen für die Anwendung sind genau wie für die `write()`-Operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{Read}\;\rule{25em}{0pt}}}\\
\Delta \mathit{SocketStateSpace}\\
p? : \mathit{PID}\\
d? : \mathit{FD}\\
\mathit{lbuffer}? : \mathbb{N}\\
\mathit{buffer}! : \mathrm{seq}\, \mathit{BYTE}\\
\mathit{bytesRead}! : \mathbb{N}\\
\hline
d? \in \mathit{fd}(p?)\\
(\exists\, a : \mathrm{dom}\ \mathit{ep} \bullet \mathit{sa}(a) = (p?, d?))\\
(\mathbf{let}\ \mathit{fid} == \mathit{fp}(p?, d?) \bullet\\
\quad \mathit{buffer}! = \mathbf{if}\ \mathit{lbuffer}? \geq \#(f(\mathit{fid}))\\
\qquad\qquad \mathbf{then}\ f(\mathit{fid})\ \mathbf{else}\ (1 \mathinner{\ldotp\ldotp} \mathit{lbuffer}?) \lhd f(\mathit{fid}) \wedge\\
\quad \mathit{bytesRead}! = \mathbf{if}\ \mathit{lbuffer}? \geq \#(f(\mathit{fid}))\\
\qquad\qquad \mathbf{then}\ \#(f(\mathit{fid}))\ \mathbf{else}\ \mathit{lbuffer}? \wedge\\
\quad f' = f \oplus \{\mathit{fid} \mapsto (\mathbf{if}\ \mathit{lbuffer}? \geq \#(f(\mathit{fid}))\\
\qquad\qquad \mathbf{then}\ \langle\,\rangle\ \mathbf{else}\ \mathit{squash}\ ((1 \mathinner{\ldotp\ldotp} \mathit{lbuffer}?) \ntrianglelefteq f(\mathit{fid})))\})\\
\mathit{fd}' = \mathit{fd}\\
\mathit{fp}' = \mathit{fp}\\
\mathit{sa}' = \mathit{sa}\\
\mathit{bl}' = \mathit{bl}\\
\hline
\end{array}
$$

Die Wirkung von *Read* ist wie folgt:

- Wenn die Länge *lbuffer*? des zur Verfügung gestellten Puffers *buffer*! größer gleich der Länge der im Socket wartenden Daten ist, wird die gesamte dem Socket zugeordnete Datei geleert und in den Puffer übertragen.
- Wenn die Länge *lbuffer*? kleiner gleich der Länge der im Socket wartenden Daten ist, werden die ersten *lbuffer*? Bytes nach *buffer*! übertragen, und in der dem Socket zugeordneten Datei verbleiben die restlichen Bytes.
- In jedem Fall wird in *bytesRead*! die Anzahl der nach *buffer*! übertragenen Bytes zurückgegeben.