Michael Stapelberg

Indifferente Socken

Programmieren für IPv4 und IPv6

Wenn der Admin eines Tages aus seinem Serverraum auftaucht und endlich IPv6 eingeführt hat, kann der Programmierer ihm fröhlich entgegenschmettern: „Bin schon da!“ Denn wenn er jetzt nur eine Hand voll Zeilen in seinem Code ändert, nutzt er vollautomatisch IPv4 oder IPv6 – was halt gerade zur Verfügung steht.

Vor mehr als zehn Jahren wurde IPv6 spezifiziert. Doch heutzutage trifft man immer noch auf Programme, die nicht mit dem neuen Protokoll umgehen können. Dabei gibt es selbst in altmodischen Hochsprachen wie C angenehme Schnittstellen, um protokollunabhängig mit Netzwerkverbindungen umzugehen. Für den Programmierer wird es dadurch teilweise sogar einfacher, denn um Details wie das Erstellen von Sockets muss er sich nicht mehr kümmern.

Eine IP-Verbindung aufzubauen ist ein klassischer Programmierer-Dreisprung: Zuerst besorgt man sich die numerische IP-Adresse zum gewünschten Host-Namen, dann konstruiert man ein Socket mit den nötigen Eigenschaften und schließlich stellt man damit die Verbindung her. Anschließend verhält sich das Socket ähnlich wie eine geöffnete Datei mit Lese- und Schreibzugriffen. Das Prinzip bleibt auch in einem Programm erhalten, das IPv4- und IPv6-Verbindungen abwickelt. Nur der erste der drei Sprünge funktioniert anders, der zweite wird sogar einfacher. Dabei ist für den Programmierer komplett transparent, um welches Protokoll (IPv4 oder IPv6) es sich handelt.

Früher war es gute Praxis, die POSIX-Funktion `gethostbyname` zur Namensauflösung zu benutzen. Sie liefert eine 32-Bit-Zahl zurück. Das ist nicht nur zu wenig für eine 128 Bit lange IPv6-Adresse, sondern entspricht auch dem überholten Prinzip, dass zu einem Namen nur eine IP-Adresse gehört. Ihr IPv6-taugliches Pendant heißt `getaddrinfo` und leistet mehr: Das Ergebnis ist eine Liste aller Adressen zum Namen.

Diese Liste sortiert das Betriebssystem geschickt vor. Vorne stehen die IPv6-Adressen – vorausgesetzt, der Computer verfügt über eine native IPv6-Verbindung. Sofern nur ein minderwertiger Tunnel per Teredo [1] oder 6to4 bereitsteht, werden weiterhin IPv4-Adressen zuerst zurückgegeben. Das soll verhindern, dass Timeouts beim Verbinden via IPv6 die Anwendung ausbremsen. Das komplette Verhalten zur Sortierung beschreibt der RFC 3484. Auf Linux-Systemen lässt es sich in der Datei `/etc/gai.conf` konfigurieren.

Weiterhin beherrscht die Funktion `getaddrinfo` Internationalized Domain Names (IDN) und bereitet die Parameter für die anschließenden Aufrufe von `socket` und `connect` vor. Daher sind die einzelnen Knoten der Rückgabe-Liste vom Typ `struct addrinfo`. Anstatt selbst eine `struct sockaddr` zu befüllen und die Portnummer zwischen Host- und Network-Byteorder zu konvertieren, übergibt man einfach direkt das Ergebnis der Funktion an `socket` und `connect` .

Das Listing rechts gibt ein Beispiel für den Verbindungsaufbau in C. Die `#include`-Orgie am Anfang, per `#ifdef` nur auf manchen Betriebssystemen zu kompilierenden Teile und einige Details sparen wir uns hier. Die vollständigen Listings finden Sie über den c't-Link am Ende des Artikels zum Download.

Die eingangs deklarierten hints enthalten Hinweise, welche Ergebnisse getaddrinfo liefern soll. Sie haben denselben Datentyp wie die Knoten der Ergebnisliste, enthalten also eine Art Vorlage. Darin legt man zum Beispiel den Socket-Typ (SOCK_STREAM für TCP oder SOCK_DGRAM für UDP) fest. TCP muss im Feld ai_protocol nicht zusätzlich angegeben werden, das versteht sich bei SOCK_STREAM von selbst. Andererseits genügt es nicht, nur ai_protocol zu setzen. Auf manchen Systemen ergibt das unbrauchbare Sockets mit falschem Typ.

Flaggenzeichen

Soll das Socket nur für IPv4 oder IPv6 taugen, fordert man dies im Parameter ai_family an; das Beispiel macht mit PF_UNSPEC keine Vorgabe. Weitere Details bestellt der Programmierer im Bit-Feld ai_flags. Ist darin über die Konstante AI_ADDRCONFIG ein Bit gesetzt, liefert getaddrinfo nur Adressen, die prinzipiell erreichbar sind. Wenn beispielsweise der lokale Rechner keine globale IPv6-Adresse hat, löscht getaddrinfo mit AI_ADDRCONFIG alle IPv6-Adressen aus der Ergebnisliste. Das funktioniert allerdings nicht immer zuverlässig. So streicht Windows gelegentlich die IPv6-Adressen, obwohl sie erreichbar wären.

Außerdem gibt es noch verschiedene Flags, die vor allem für die Serverprogrammierung relevant sind. Für Client-Programme ist gegebenenfalls das Flag AI_NUMERICHOST interessant, welches angibt, dass getaddrinfo nur numerische Adressen akzeptiert, damit es keine Netzzugriffe verursacht, um den Hostnamen aufzulösen. Auf diesem Weg kann man verifizieren, ob ein String tatsächlich eine gültige IP-Adresse enthält.

Zwischen dem Namen und den hints nimmt getaddrinfo auch eine Port-Angabe für TCP oder UDP entgegen. In diesem String darf entweder direkt die Portnummer stehen oder ein Name, dem in der Datei /etc/services (unter Windows %windir%\System32\drivers\etc\services) eine Nummer zugewiesen ist.

Schlägt getaddrinfo fehl, liefert es einen von Null verschiedenen Fehlercode. Den übersetzt die ungemein praktische Funktion gai_strerror in eine Fehlermeldung in der Landessprache.

Damit ist der erste Sprung gelandet und es geht ans Öffnen der Verbindung. Bei getaddrinfo muss man immer mit mehreren Ergebnissen rechnen, die eventuell sogar unterschiedliche Adressfamilien benutzen. Auch ohne IPv6 gehören zu vielen Namen mehrere Adressen, wie zum Beispiel nslookup www.google.com zeigt (Round-Robin-DNS). Die Anbieter verteilen damit die Last und liefern Alternativen für den Fall, dass mal einer der Server nicht erreichbar ist.

Um von einer schlechten Idee gleich abzuraten: Versuchen Sie nicht, alle Adressen der Liste gleichzeitig anzusprechen, und dann die zuerst aufgebaute Verbindung zu benutzen. Üblicherweise führt das nicht schneller zu einer Verbindung, da getaddrinfo bereits gut sortiert und somit der erste Verbindungsversuch meistens funktioniert. Dem steht erheblicher Verwaltungsaufwand für Threads oder Prozesse gegenüber, die für den parallelen Aufbau nötig sind. Weiterhin verursacht der Versuch viel Netzauslastung und Serverlast, denn wenn man zehn Verbindungen aufbaut, müssen die neun unnötigen auch auf dem Server wieder geschlossen werden.

Die for-Schleife iteriert also über das Ergebnis und versucht, ein Socket anzulegen und zu verbinden. Zu beachten ist hierbei, dass im Ergebnis von getaddrinfo Adressfamilie, Socket-Typ, Protokoll und insbesondere die Adresse bereits vorbereitet sind. Es reicht also völlig aus, diese direkt an die socket- und connect-Funktionen zu übergeben. Sofern einer der Aufrufe fehlschlägt, geht das continue auf den nächsten Eintrag der Ergebnisliste über. Hat alles geklappt, bricht break die Schleife ab.

Ein Fehler tritt bei connect üblicherweise dann auf, wenn der Server die Verbindung ablehnt (connection refused), es eine Zeitüberschreitung bei der Verbindung (timeout) oder ein anderes Netzwerkproblem gibt (zum Beispiel no route to host, network unreachable). Die Fehlerbehandlung für den socket-Befehl braucht man unter anderem für Systeme, die gar keinen IPv6-Support im Kernel haben und deshalb an IPv6-Einträgen in der Ergebnisliste scheitern.

Fiese Fehler

Das Beispiel verzichtet absichtlich auf Fehlermeldungen; einerseits um Platz zu sparen, andererseits, weil das sinnvolle Maß an Meldungen von der Nutzerschaft abhängt. Hier müssen Sie sich überlegen, welche Fehlermeldungen Sie in welcher Art und Weise ausgeben möchten. Ist jeder Fehler beim Verbindungsaufbau relevant? Schließlich sollte der versierte Nutzer oder Administrator Bescheid wissen, wenn ein Dienst theoretisch über IPv6 erreichbar ist, die Verbindung aber fehlschlug. Andererseits wundern sich unbedarfte Benutzer, wenn das Programm Fehlermeldungen ausspuckt, obwohl es dann doch funktioniert.

Nach Ende der Schleife wird die Ergebnisliste nicht mehr benötigt und der Programmierer muss sich C-typisch selbst um die Entsorgung kümmern. Die Funktion freeaddrinfo räumt die Liste auf einen Schlag aus dem Speicher.

An dieser Stelle ist sock -1, wenn alle Verbindungsversuche fehlgeschlagen sind, oder es enthält den File Descriptor eines verbundenen Sockets. Zum Schreiben und Lesen unterscheidet es sich nicht von seinem auf die alte Weise erzeugten reinen IPv4-Pendant. Wer seinen alten Code IPv6-fähig machen möchte, muss von dieser Stelle an in der Regel kaum noch etwas ändern.

Das Beispielprogramm benutzt das Socket nicht, sondern zeigt nur an, mit welcher Adresse es verbunden ist. Ganz klassisch dient die Funktion mit dem etwas irreführenden Namen getpeername dazu, Adresse und Port der Gegenstelle eines verbundenen Sockets abzufragen. Sie erwartet seit jeher einen Zeiger auf ein struct sockaddr, um es mit der Antwort zu befüllen. Doch dieser Datentyp ist nur noch ein Platzhalter, weil je nach Adressfamilie unterschiedlich große Strukturen nötig sind – die ursprünglichen 16 Bytes reichen gerade mal für eine IPv6-Adresse ohne Typ-Angabe, Port und Protokoll. Daher gibt es den Typ struct sockaddr_storage, der auf jeden Fall für alle Adressen groß genug ist. Die Definition der Variable sa dient allein der Lesbarkeit des Codes, damit nicht in jedem Funktionsaufruf wieder die sperrige Typumwandlung stehen muss.

Damit der Programmierer keine umständlichen Fallunterscheidungen klöppeln muss, um die struct sockaddr in eine lesbare Form zu bringen, gibt es die Funktion getnameinfo. Sie zerlegt eine struct sockaddr in Host und – falls vorhanden – Port. Über Bits im Parameter flags steuert man ihr Verhalten. So liefert NI_NUMERICHOST | NI_NUMERICSERV anstelle der Host- und Port-Namen deren numerische Darstellung. Sonst würde die Funktion im DNS den kanonischen Namen des Hosts

```
int main() {
  struct addrinfo hints;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  struct addrinfo *res;
  int res;
  if (ret = getaddrinfo("www.example.com",
    "http", &hints, &res)){
    fprintf(stderr, "getaddrinfo: %s\n",
      gai_strerror(ret));
    return 1;
  }

  int sock = -1;
  struct addrinfo *walk;
  for (walk = res; walk != NULL;
    walk = walk->ai_next) {
    sock = socket(walk->ai_family,
      walk->ai_socktype, walk->ai_protocol);
    if (sock < 0)
      continue;
    if (connect(sock, walk->ai_addr,
      walk->ai_addrlen) ) {
      /* Ergebnis !=0, das ging schief */
      /* Vorsichtshalber Socket schließen */
      close(sock);
      sock = -1;
      continue;
    }
    break;
  }

  freeaddrinfo(res);
  if (sock == -1)
    return 1;

  struct sockaddr_storage sa_stor;
  socklen_t sas = sizeof(sa_stor);
  struct sockaddr* sa =
    (struct sockaddr*) &sa_stor;
  if(getpeername(sock, sa, &sas))
    return 1;

  char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
  int flags = NI_NUMERICHOST | NI_NUMERICSERV;
  if(getnameinfo(sa, sas, hbuf, sizeof(hbuf),
    sbuf, sizeof(sbuf), flags))
    return 1;

  printf("Verbunden mit Host %s, Port %s\n",
    hbuf, sbuf);
}
```

Der Aufbau einer IP-Protokoll-unabhängigen TCP-Verbindung unterscheidet sich kaum von der reinen IPv4-Variante.

```
int main() {
  struct addrinfo *res, hints;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  if(int ret = getaddrinfo(NULL, "49090",
    &hints, &res)){
    fprintf(stderr,"getaddrinfo(): %s\n",
      gai_strerror(ret));
    return 1;
  }

  int fd = 0, n = 0;
  #define NFDS 2
  int sockfds[NFDS];

  struct addrinfo *walk;
  for (walk = res; walk != NULL;
    walk = walk->ai_next) {
    fd = socket(walk->ai_family,
      walk->ai_socktype, walk->ai_protocol);
    if (fd == -1) continue;

    if (walk->ai_family == AF_INET6) {
      int on = 1;
      if ( -1 == setsockopt(fd, IPPROTO_IPV6,
        IPV6_V6ONLY, (char*)&on, sizeof(on)))
          continue;
    }

    if (bind(fd, walk->ai_addr,
      walk->ai_addrlen))
        continue;

    if (listen(fd, 5) == -1)
      continue;

    struct sockaddr_storage sa_stor;
    int sas = sizeof(sa_stor);
    struct sockaddr* sa =
      (struct sockaddr*) &sa_stor;
    if( getsockname(fd, sa, &sas))
      return 1;

    char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
    int flags = NI_NUMERICHOST|NI_NUMERICSERV;
    if( getnameinfo( sa, sas, hbuf,
      sizeof(hbuf), sbuf, sizeof(sbuf), flags))
       return 1;

    printf("Lausche an %s, Port %s (fd %d)\n",
      hbuf, sbuf, fd);
    sockfds[n++] = fd;
    if(n>=NFDS)
      break;
  }
  freeaddrinfo(res);

  fd_set fds;
  while (1) {
    int highest = 0;
    FD_ZERO(&fds);
    for (int i = 0; i < n; i++) {
      FD_SET(sockfds[i], &fds);
      highest = (sockfds[i] > highest ?
        sockfds[i] : highest);
    }
    ret = select(highest+1, &fds, NULL, NULL,
      NULL);
    if (ret == -1) return 1;

    for (int i = 0; i < n; i++) {
      if (!FD_ISSET(sockfds[i], &fds))
        continue;

      struct sockaddr_storage peer;
      socklen_t peers = sizeof(peer);
      int cfd = accept(sockfds[i],
        (struct sockaddr*)&peer, &peers);

      char buf[NI_MAXHOST];
      if (getnameinfo( (struct sockaddr*)&peer,
        peers, buf, sizeof(buf), NULL, 0,
        NI_NUMERICHOST))
          return 1;

      printf("Verbindung auf fd %d von %s\n",
        sockfds[i], buf);
      close(cfd);
    }
  }
}
```

Ein Server sollte immer IPv4- und IPv6-Sockets getrennt behandeln.

nachfragen und aus /etc/services den Dienstnamen holen.

Server

Wenn ein reiner IPv4-Server alle Schnittstellen bedienen soll, bindet man mit bind ein Socket an die Adresse 0.0.0.0 [2]. In Dual-Stack-Programmen gibt es dafür zwei Methoden: Ein einzelnes IPv6-Socket gebunden an die Adresse :: im „Hybridmodus“ akzeptiert auch IPv4-Verbindungen. Diese werden dann als „Mapped Addresses“ dargestellt, also zum Beispiel ::ffff:192.0.2.1. Allerdings funktioniert das nicht auf allen Betriebssystemen, weswegen man von der Verwendung absehen sollte. Auch wenn Portabilität keine Rolle spielt, ist dies kein guter Ansatz, denn es ist grundsätzlich umständlich, mit Mapped Addresses zu arbeiten. Damit das eigene Programm definitiv keine solchen Sockets nutzt, muss man die Hybrid-Option abschalten, mit der Socket-Option IPV6_V6ONLY (mehr dazu gleich).

Die zweite Methode ist, getaddrinfo mit NULL als Hostnamen aufzurufen. Dann enthält die Ergebnisliste die allgemeine Adresse für alle Protokoll-Familien, also mindestens 0.0.0.0 und ::. Und schon hat man mehrere Adressen und damit mehrere Sockets. Insbesondere beim Portieren von Anwendungen entsteht also etwas Mehraufwand dadurch, dass mehrere Sockets zu bedienen sind.

Im Server-Beispiel links sehen die ersten Zeilen fast genauso aus wie beim Client. Nur das Flag AI_PASSIVE bestimmt, dass getaddrinfo Adressen ermitteln soll, an die sich ein Server-Socket binden kann. Mit diesen Parametern liefert die Funktion derzeit höchstens zwei Adressen, eine für IPv4 und eine für IPv6. Daher reicht für die Verwaltung der Server-Sockets ein Array mit wenigen Plätzen. Um auf alles vorbereitet zu sein, könnte man auch eine dynamische Liste benutzen.

Die Verarbeitung der Adressen läuft ähnlich wie im Client: Sofern eine Socket-Operation fehlschlägt (etwa weil dieser Port bereits von einem anderen Serverprogramm belegt ist), geht es mit der nächsten Adresse weiter. Für alle IPv6-Sockets (erkennbar an der ai_family AF_INET6) wird dann die erwähnte Option IPV6_V6ONLY eingeschaltet, damit sie nicht im Hybridmodus versehentlich IPv4-Verbindungen annehmen. Dies ist die einzige IPv6-spezifische Stelle im gesamten Code.

Statt connect kommen beim Server bind und listen zum Einsatz [2]. Falls alles gut geht, gibt das Programm die Adresse des Sockets aus. An Stelle von getpeername benutzt es dabei getsockname für das eigene Ende des Sockets, sonst läuft alles wie beim Client. Schließlich landet das fertige Socket im Array.

Am Ende der Schleife enthält sockfds die fertigen Sockets, die bereit sind fürs Verbinden. Der Server muss sie nun alle beobachten, denn wenn ein Client anklopft, muss der Server die Verbindung mit accept annehmen. Eine Methode ist, pro Socket einen eigenen Prozess oder Thread anzuwerfen. Das ist bei High-Performance-Servern sinnvoll, die sehr viele Verbindungen bedienen müssen. Es führt allerdings zu Verwaltungsaufwand beim Beenden des Servers. Unser Beispiel nutzt daher select, um die Arbeit ans Betriebssystem auszulagern.

Multiselection

Der etwas kompliziert aussehende Code tut nichts anderes, als in einem Bit-Feld vom Typ fd_set für jedes Socket das zugehörige Bit zu setzen. Dabei merkt er sich in highest, welches das höchstwertige Bit ist. Der Aufruf von select veranlasst das Betriebssystem, die ausgewählten Sockets zu beobachten. Im letzten Parameter steht ein Timeout; der Wert NULL bedeutet hier unendlich. Das Programm verharrt daher beim select-Aufruf, bis es auf irgendeinem der Sockets etwas zu lesen gibt. In den beiden weiteren Parametern hätte man Sockets übergeben können, die auf Schreibbereitschaft geprüft beziehungsweise ignoriert werden.

Wenn es nach dem select weitergeht, sind im Bit-Feld nur noch die Bits derjenigen Sockets gesetzt, auf denen sich etwas getan hat. Danach durchsucht das Programm das Feld und nimmt mit accept die Verbindung an. Die dabei übergebene Adresse der Gegenstelle wandelt es wie gewohnt per getnameinfo um und zeigt sie an. Dann schließt es das beim accept kopierte Socket und wartet auf die nächste Aktivität.

Um den Server zu testen, tippen Sie im Browser Adressen wie http://127.0.0.1:49090/, http://[::1]:49090/ oder http://localhost:49090/ ein. Der Browser zeigt zwar eine Fehlermeldung, aber der Server sollte Verbindungen vermelden.

Unser Beispiel bindet sich der Einfachheit halber wahllos an alle Adressen. Doch wenn man ohnehin mehrere Sockets bedient, kann man das auch gleich richtig machen, nämlich mit konfigurierbaren Adressen. Denn IPv6-taugliche Hosts haben normalerweise einen ganzen Sack voll Adressen, die sich teilweise schnell ändern. Das Parsen einer Konfigurationsdatei würde weit über diesen Artikel hinausgehen. Doch als dessen Ergebnis darf man ein Array oder besser eine Liste erwarten, die pro Knoten eine Adresse und eine Port-Nummer enthält. Die geht man in einer zusätzlich außenherum gebauten Schleife durch und verfüttert sie jeweils an getaddrinfo. Die fertigen Sockets steckt man wie im Beispiel in ein Array oder wiederum in eine Liste. Die Behandlung dieser Sockets unterscheidet sich dann nicht von der im Beispiel: Bit-Feld füllen, auf select warten und die neue Verbindung bedienen. (je)

Literatur

[1] Johannes Endres, Reiko Kaps, Teredo bohrt IPv6-Tunnel durch Firewalls, http://heise.de/-221537

[2] Johannes Endres, Bedienung!, Netzwerk-Server selbst programmiert, c’t 5/04, S. 206

www.ct.de/1019160

Windows-Wunderlichkeit

Unsere Experimente deckten einen Fehler in der Windows-Implementierung von getaddrinfo auf: Sie ignoriert das AI_ADDRCONFIG-Bit und verhält sich immer so, als wäre es gesetzt. Die Ergebnisliste enthält also nur die Zieladressen, von denen Windows glaubt, dass es sie erreichen kann. Leider geht das System dabei sehr konservativ vor. Dass zum Beispiel die IPv6-Verbindung am Router zusammengebrochen ist, merkt es sehr schnell und verwirft IPv6-Adressen. Um zu bemerken, dass die Verbindung wieder da ist, braucht gelegentlich sogar Windows 7 einen Neustart. Das betrifft alle Programme, die sich auf getaddrinfo verlassen, also beispielsweise Firefox und alle .NET-Programme, die Klassen wie TCPClient verwenden.

Für Programme, die getaddrinfo nutzen, um anschließend eine Verbindung aufzubauen, stellt der Bug kaum ein Problem dar, weil in der Regel die IPv4-Verbindung klappt. Doch eigentlich sollte die Funktion auch dazu taugen, auf einem Rechner ohne IPv6-Interface die IPv6-Adresse eines anderen Rechners nachzusehen. Durch den Bug klappt das nicht.

Mac-Macke

Untersuchungen von anderen IPv6-Experten haben gezeigt, dass Mac OS X die Ergebnisliste nicht so wie die anderen Systeme sortiert, sondern auch die weniger stabilen 6to4-Tunnel gegenüber IPv4 bevorzugt. Wenn der Mac hinter einem der üblichen IPv4-NAT-Router steht, entspricht das sogar einer möglichen Auslegung des RFC 3484, führt jedoch oft zu schlechten Verbindungen.

Bei Mac OS X 10.6 (Snow Leopard) erscheinen zwar alle Einträge doppelt in der Ergebnisliste, dafür fehlen gelegentlich einige. Eine systematische Ursache konnten wir nicht finden.

Perl-Patzer

Anders als das obsolete gethostbyname gehört getaddrinfo nicht zu den internen Funktionen von Perl. Sie versteckt sich im Modul Socket::GetAddrInfo, das kein Standardmodul ist. Die Nachinstallation klappt nur, wenn auf dem Rechner ein C-Compiler so installiert ist, dass der Perl-Paketmanager ihn auch benutzt. Sonst installiert er – ohne Fehlermeldung! – eine Ersatzversion des Moduls, das keinerlei IPv6-Funktion hat.

Java-Jammer

Java enthält die Klasse Socket, die nur einen Namen und den Port braucht, um eine TCP-Verbindung herzustellen. Zur Namensauflösung benutzt sie die Methode getAllByName der Klasse InetAddress, die sich grundsätzlich nicht überreden lässt, ihre Ergebnisliste gemäß RFC 3484 zu sortieren. In der Grundeinstellung stellt Java IPv4-Adressen an den Anfang der Liste. Setzt man die Property java.net.preferIPv6Addresses auf true, stehen immer die IPv6-Adressen vorne. Das sollte man jedoch auf jeden Fall vermeiden, denn die Socket-Klasse probiert es immer nur bei der ersten Adresse der Liste. Ist das auf einem Rechner ohne IPv6 eine IPv6-Adresse, endet der Versuch also mit einer Fehlermeldung, selbst wenn eine IPv4-Verbindung möglich wäre. De facto benutzen als Java-Clients nur dann IPv6, wenn der Verbindungspartner ausschließlich diese Protokoll spricht.

c't