Portierung von Anwendungen nach IPv6

Tips & Tricks

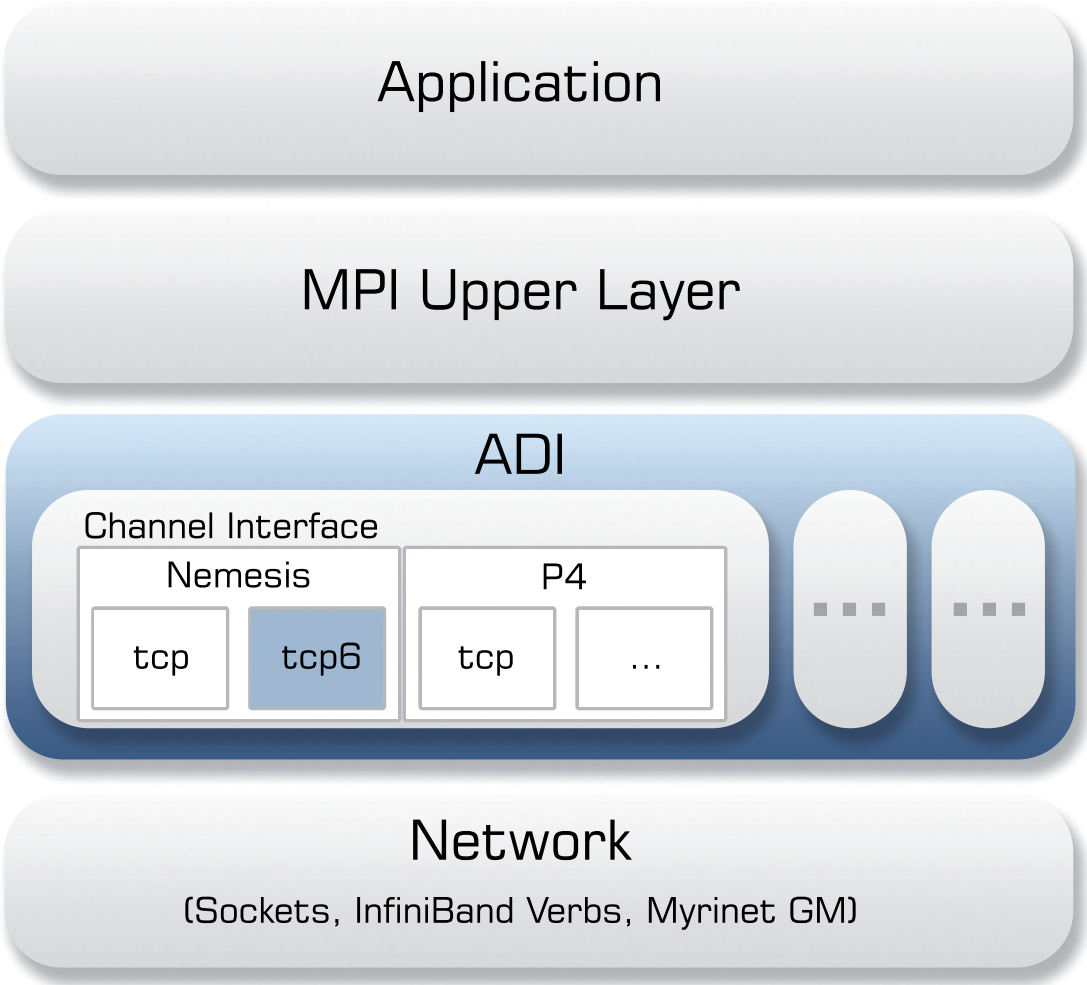
Oliver Eggert

Universität Potsdam
Institut für Informatik
Professur Betriebssysteme und Verteilte Systeme

Potsdam, den 05. November 2011

Gliederung

Beispiel: MPICH2



→ weitere IPv6-Projekte: `http://www.ipv6-showcase.de/`

Beispiel: IPv6 Intrusion Detection Systems

Honeyd v6

- simuliert virtuelle Netzwerke
- betriebssystemspezifisches Verhalten
- Angriffserkennung und -bewertung

Snort IPv6 Preprocessor

- Intrusion Detection System
- analysiert Netzwerkpakete (live)
- bei Angriffen werden Pakete verworfen

→ siehe `http://www.ipv6-ids.de/`

Beispiel: IPv6 Intrusion Detection Systems

Honeyd v6

- simuliert virtuelle Netzwerke
- betriebssystemspezifisches Verhalten
- Angriffserkennung und -bewertung

Snort IPv6 Preprocessor

- Intrusion Detection System
- analysiert Netzwerkpakete (live)
- bei Angriffen werden Pakete verworfen

→ siehe `http://www.ipv6-ids.de/`

IPv6 aus Entwicklersicht

Kernfragen:

- Wie wird mit IPv6 in der jeweiligen Programmiersprache umgegangen?
- Analyse: Wie ist die vorliegende Anwendung aufgebaut?
- Was sollte beim Portieren der Anwendung nach IPv6 beachtet werden?
- Was sollte beim Design einer neuer Anwendung beachtet werden?

IPv6 in verschiedenen Sprachen

Moderne (Hoch-)Sprachen (*dritte Generation*) behandeln IPv4/IPv6 auf der Netzwerkschicht oftmals weitestgehend transparent für den Entwickler:

- Java: `class InetAddress`
- Python: `import socket`
- Perl: `use Net::IP;` (in CPAN, nicht im Core)
- ...

Java – Beispiel

```
public static void main(String[] args) {
  try {
    InetAddress addr = null;
    addr = InetAddress.getByName("www.heise.de");
    System.out.println(addr);

    addr = InetAddress.getByName("six.heise.de");
    System.out.println(addr);

    /* liefert:
    * www.heise.de/193.99.144.85
    * six.heise.de/2a02:2e0:3fe:100:0:0:0:6
    */

    Socket s = new Socket(addr, 80);
  } catch (Exception e) { e.printStackTrace(); }
}
```

Python – Beispiel

```
import socket

s = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
s.connect("[2a02:2e0:3fe:100:0:0:0:6]", 80);

# Namensaufloesung per socket.getaddrinfo
```

Perl – Beispiel

```
$ip = new Net::IP ('193.0.1.46')

$ip = new Net::IP ('195.114.80/24')

$ip = new Net::IP ('20.34.101.207 - 201.3.9.99')

$ip = new Net::IP ('dead:beef::/32')

$ip = new Net::IP ('195/8',4);

# Namensaufloesung mit Net::DNS::Resolver,
# oder selbst query mit `nslookup -q=AAAA six.heise.de`
```

IPv6 in verschiedenen Sprachen

Moderne (Hoch-)Sprachen (*dritte Generation*) behandeln IPv4/IPv6 auf der Netzwerkschicht oftmals weitestgehend transparent für den Entwickler:

- Java: `class InetAddress`
- Python: `import socket`
- Perl: `use Net::IP;` (in CPAN nicht im Core)
- ...

→ ***Was ist mit C?***

IPv6 in verschiedenen Sprachen

Moderne (Hoch-)Sprachen (*dritte Generation*) behandeln IPv4/IPv6 auf der Netzwerkschicht oftmals weitestgehend transparent für den Entwickler:

- Java: `class InetAddress`
- Python: `import socket`
- Perl: `use Net::IP;` (in CPAN nicht im Core)
- ...

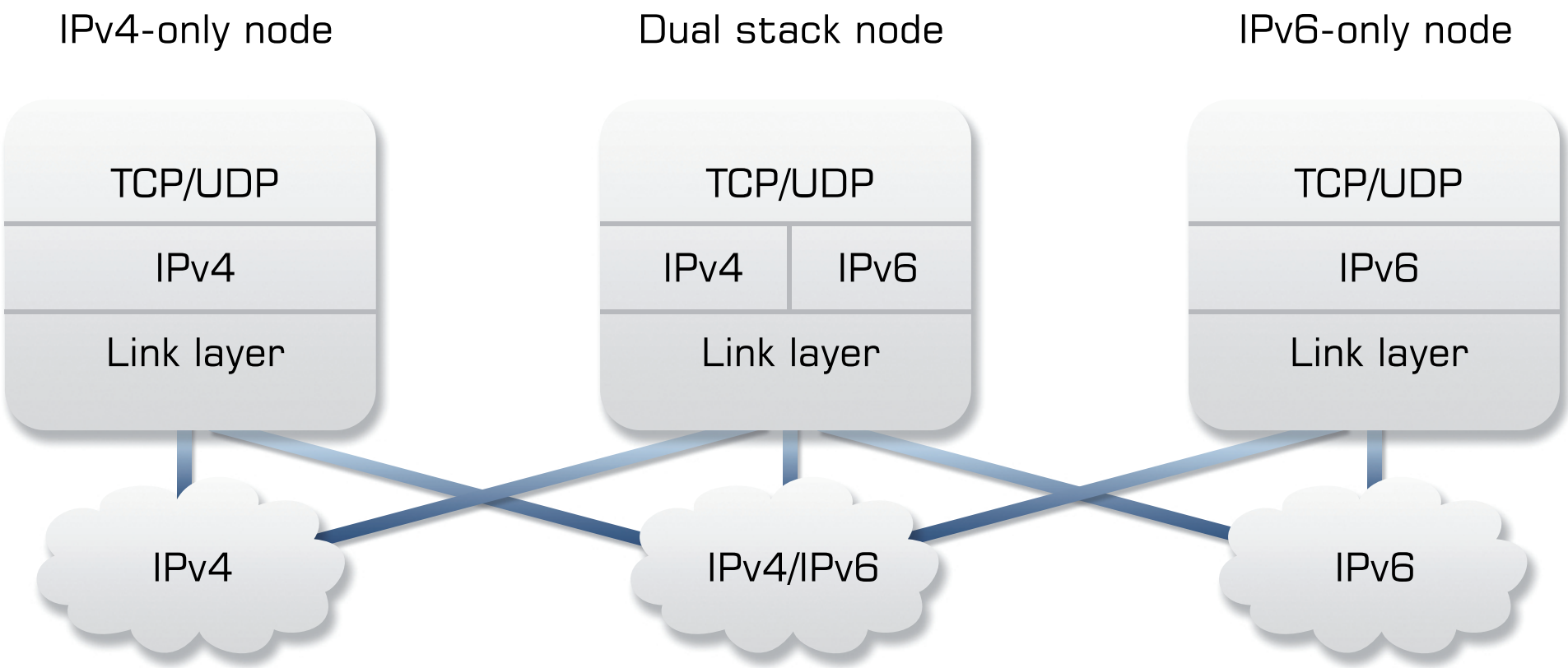
→ ***Was ist mit C?***

IPv6 in verschiedenen Anwendungen

- die meisten Anwendungen benötigen nur minimale Änderungen, um IPv6 zu unterstützen
- netzwerkorientierte Anwendungen benötigen je nach Sprache oft etwas mehr Aufwand (Bsp.: IDS, Firewall, Netzwerk-/Sicherheits-Tools, . . .)

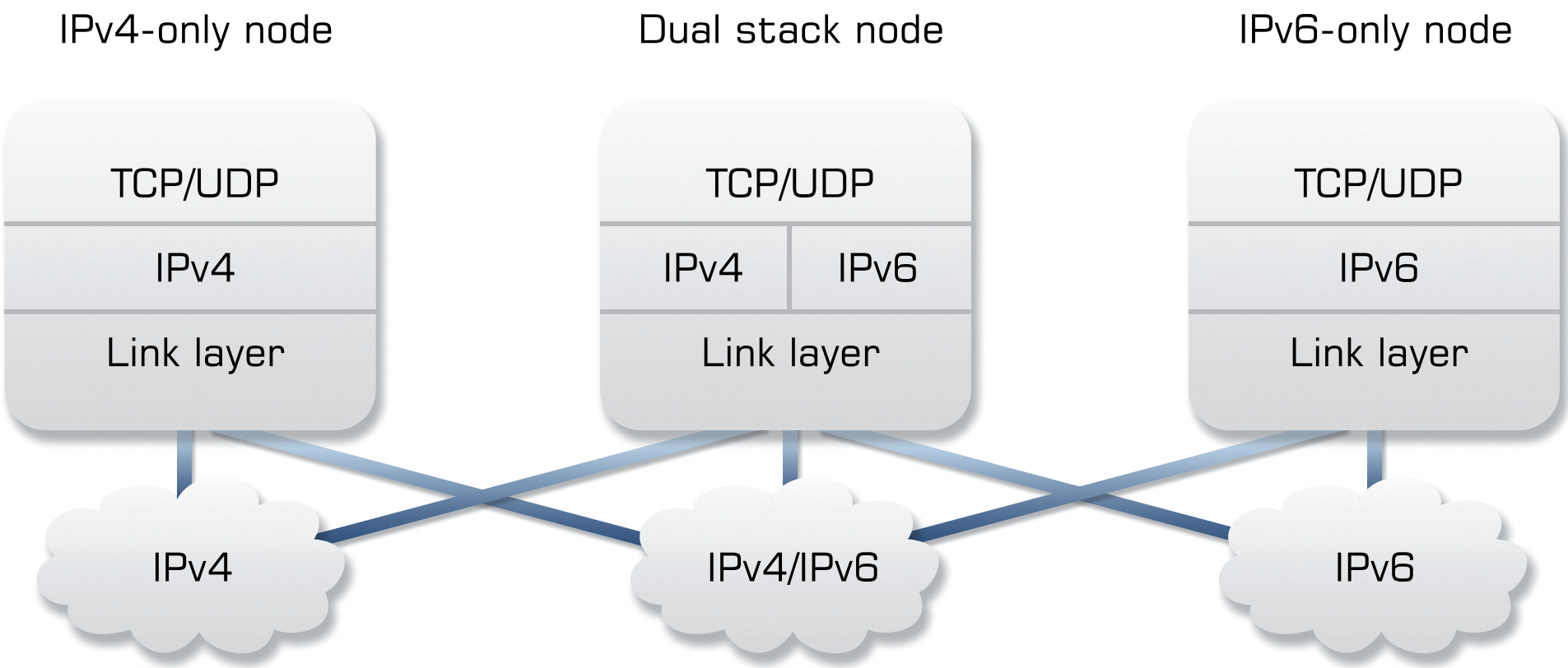
Gliederung

Stackvariationen



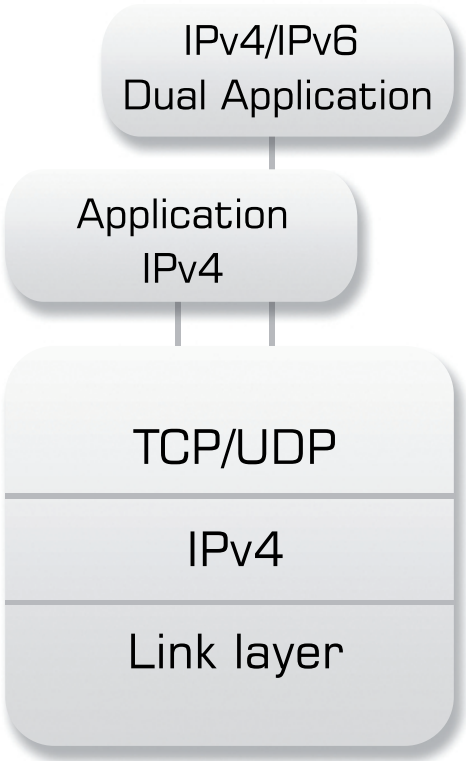
→ Dual-IPv4/IPv6-Stack in den verbreiteten Betriebssystemen

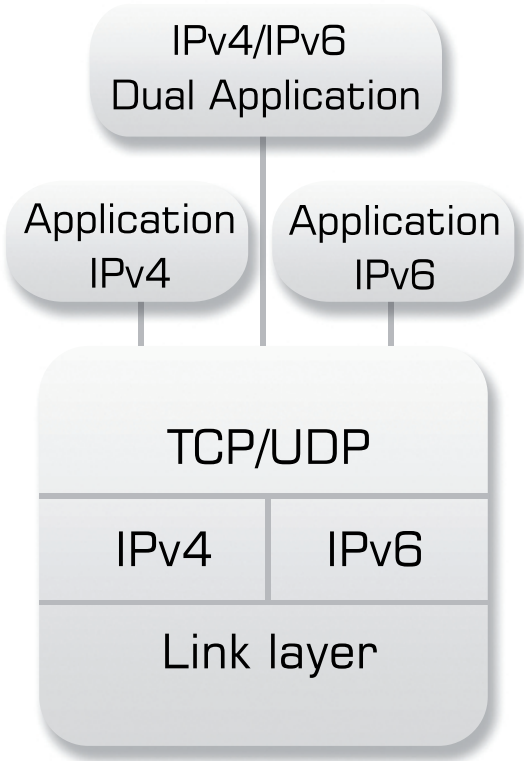
Stackvariationen

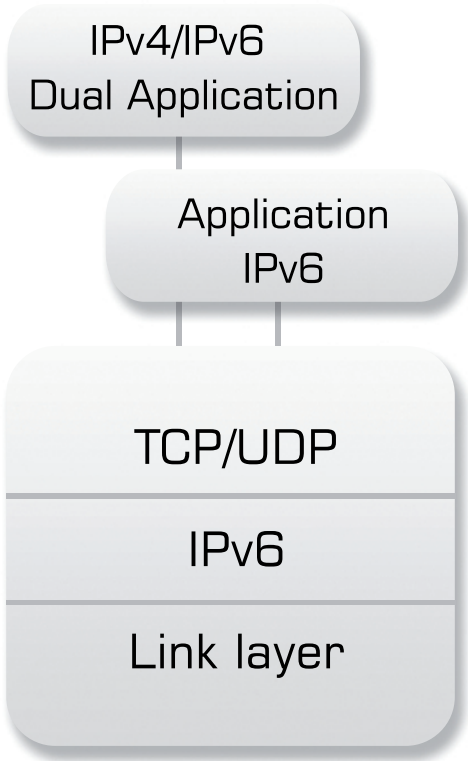


→ Dual-IPv4/IPv6-Stack in den verbreiteten Betriebssystemen

Anwendungsvariationen auf Stacks







Übergang von IPv4 zu IPv6 in Anwendungen

IPv4-Anwendung → IPv6-only

+ sehr einfaches Ersetzen von Code
- keine Rückwärtskompatibilität
- (Pflegen von) zwei Anwendungsversionen

IPv4-Anwendung → Dual IPv4/IPv6

+ Rückwärtskompatibilität
+ (Pflege) genau einer Anwendungsversion
- mehr Änderungen am Code, mehr Aufwand

→ Mehraufwand für Dual IPv4/IPv6-Anwendung lohnt sich!

Übergang von IPv4 zu IPv6 in Anwendungen

IPv4-Anwendung → IPv6-only

+ sehr einfaches Ersetzen von Code
- keine Rückwärtskompatibilität
- (Pflegen von) zwei Anwendungsversionen

IPv4-Anwendung → Dual IPv4/IPv6

+ Rückwärtskompatibilität
+ (Pflege) genau einer Anwendungsversion
- mehr Änderungen am Code, mehr Aufwand

→ Mehraufwand für Dual IPv4/IPv6-Anwendung lohnt sich!

Gliederung

String-Repräsentation von IP Adressen

Probleme

- allozierter Speicher für IPv4-String reicht u.U. nicht für IPv6-String
- IPv4 trennt Adress-Teile per “.” und IPv6 trennt per “:”
- Ports werden ebenfalls mit “:” angehängt (doppeldeutig bei IPv6)

Lösungen

- allozierten Speicher vergrößern
- Parser anpassen für Trenner und Ports
- FQDN benutzen und/oder URIs nach RFC 3986 bilden:
  `ldap://[2001:db8::7]:389/c=GB?objectClass?one`

Versionsabhängige Transport-Layer-Nutzung

Probleme

- IPv4-orientierte Datenstrukturen
- IPv4-abhängige Adressumwandlung
- versionsabhängige Socketaufrufe

Lösungen

- Nutzen von neuen, IP-Versions-unabhängigen Datenstrukturen und Adressumwandlungs-Funktionen

$\rightarrow$ Anpassung/Entwicklung von IP-Versions-unabhängigem Code

Adress- und Namensauflösung

Probleme

- fest kodierte IP-Adressen
- existierende IPv4-basierte, spezielle Adressen (`INADDR_ANY`/`INADDR_LOOPBACK`)
- existierende IPv4-basierte Adress- und Namensauflösungs-Funktionen

Lösungen

- FQDN statt fest kodierten IP-Adressen
- Anpassen der speziellen Adressen auf neue Macros
- Nutzen von neuen IP-Versions-unabhängigen Auflösungs-Funktionen

Gliederung

Generische Socket Adress-Struktur

```
struct sockaddr {
  uint8_t       sa_len;       /* implementation dependent */
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

IPv4 Socket Adress-Struktur

```
struct in_addr {
  in_addr_t    s_addr;          /* 32-bit IPv4 address */
                                /* network byte ordered */
};

struct sockaddr_in {
  uint8_t          sin_len;     /* implementation dependent */
  sa_family_t      sin_family;  /* AF_INET */
  in_port_t        sin_port;    /* 16-bit TCP or UDP port number */
                                /* network byte ordered */
  struct in_addr   sin_addr;    /* 32-bit IPv4 address */
                                /* network byte ordered */
  char             sin_zero[8]; /* unused */
};
```

IPv6 Socket Adress-Struktur

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* implementation dependent */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* interfaces set for a scope */
};
```

Protokollunabhängige Adress-Struktur

```
struct sockaddr_storage {
  uint8_t       ss_len;     /* implementation dependent */
  sa_family_t   ss_family;  /* address family: AF_xxx value */
  /*
   * implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

- kann auf protokollspezifische Adress-Strukturen gecasted werden

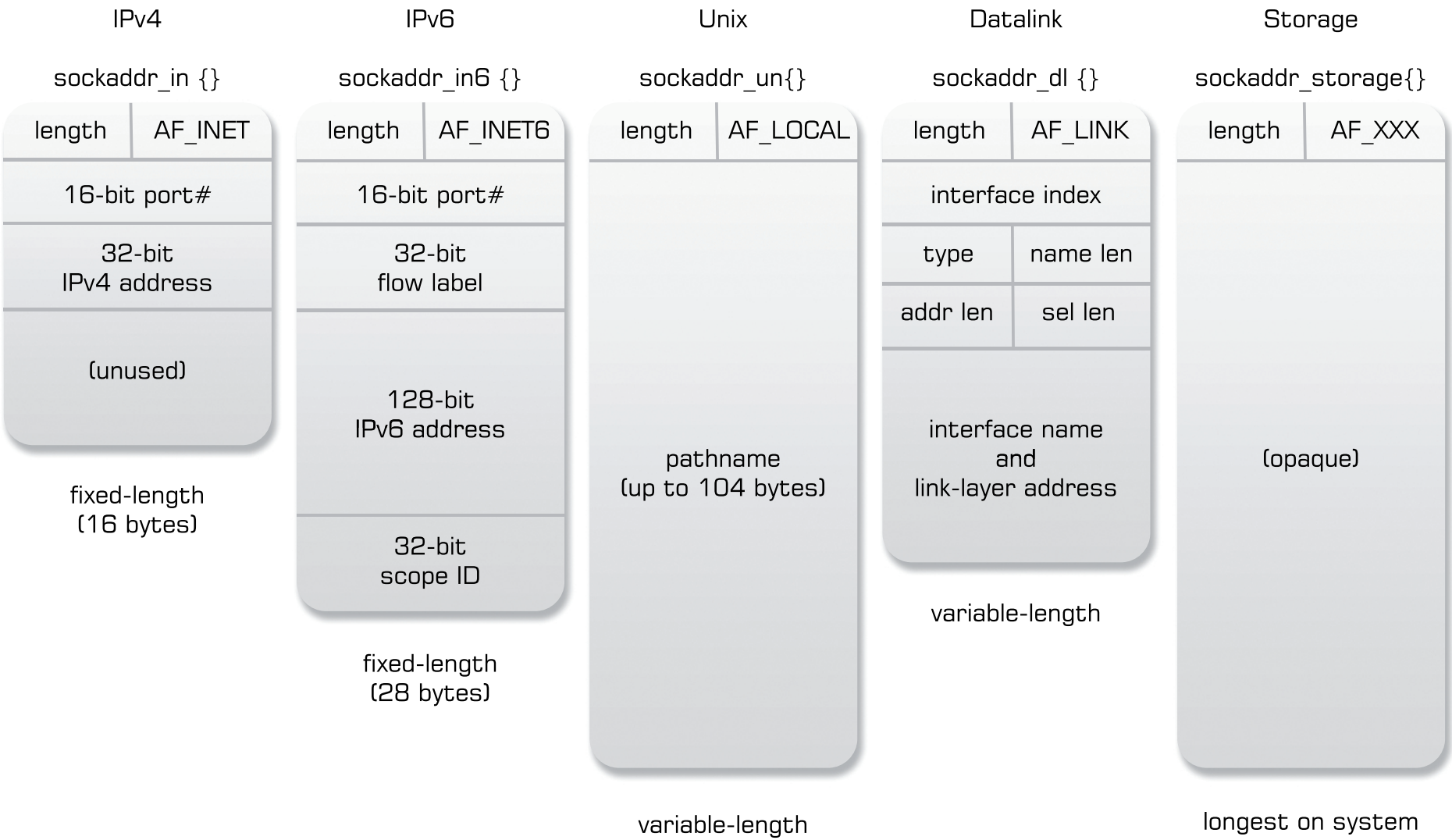
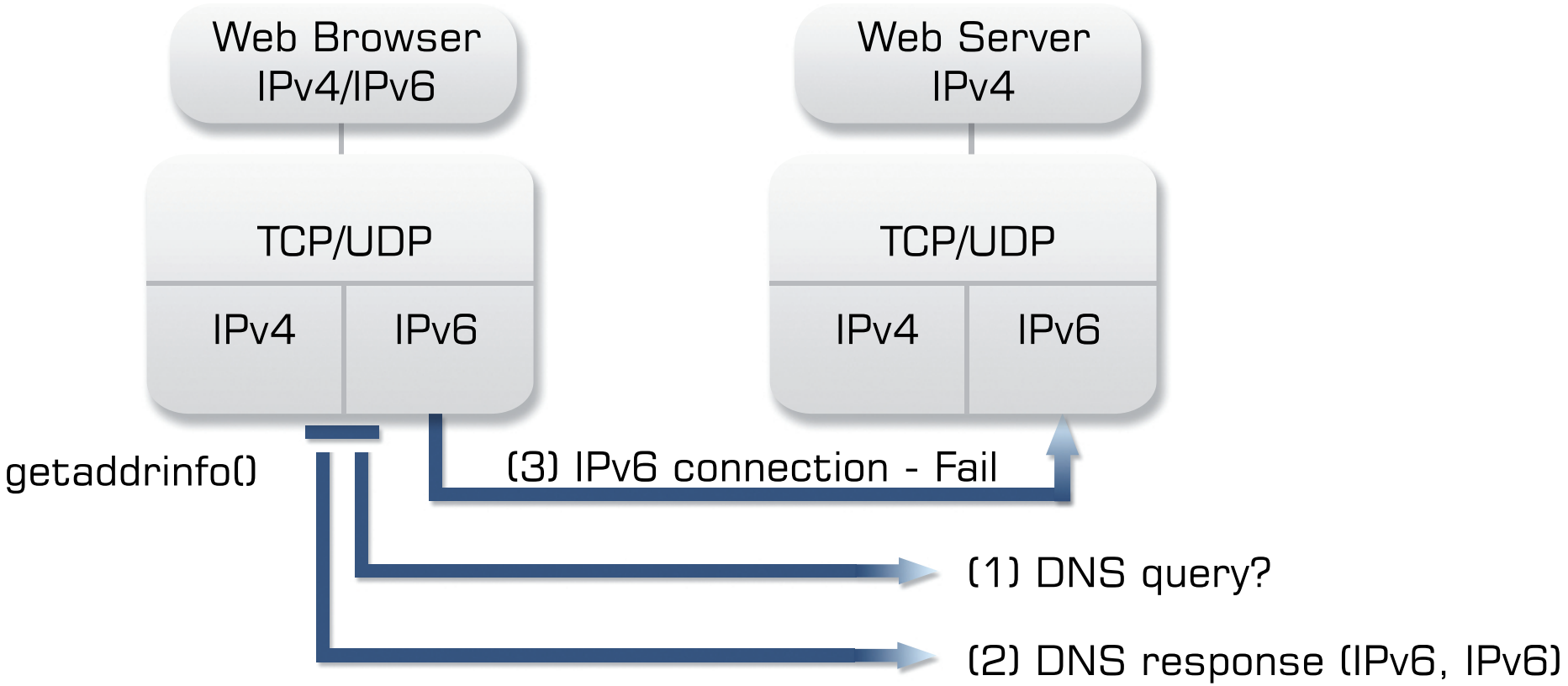


Abbildung in Anlehnung an: UNIX Network Programming Volume 1, W. Richard Stevens, Addison Wesley, 2003.

Adress-Strukturen – Beispiel

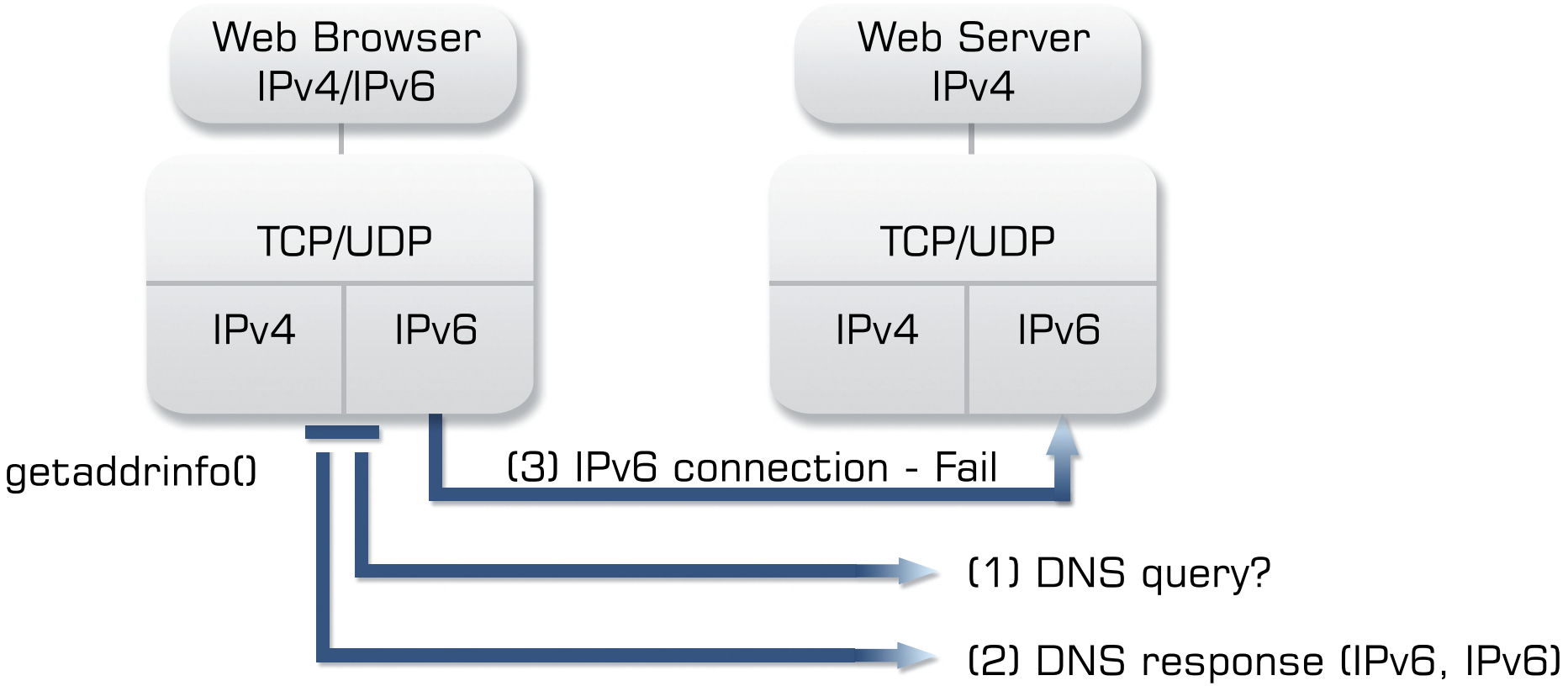
```
/* IPv4 */
struct sockaddr_in address;
accept(socket_id, (struct sockaddr *) &address, &len);


/* IPv6 */
struct sockaddr_in6 address;
accept(socket_id, (struct sockaddr *) &address, &len);


/* independent */
struct sockaddr_storage address;
accept(socket_id, (struct sockaddr *) &address, &len);
```

Anwendungen und DNS



→ keine Rückschlüsse vom DNS auf die Anwendung möglich

Anwendungen und DNS



→ keine Rückschlüsse vom DNS auf die Anwendung möglich

Adress- und Namensauflösungen

IPv4	IPv4/IPv6
`gethostbyname()`	`getaddrinfo()`
`gethostbyaddr()`	`getnameinfo()`

- Einfluss auf Rückgabewerte von `getaddrinfo()` und `getnameinfo()` über Funktions-Parameter

→ immer und überall die protokollunabhängigen Funktionen `getaddrinfo()` und `getnameinfo()` verwenden

→ Client sollte jede von `getaddrinfo()` zurückgegebene Adresse versuchen, um zu verbinden

→ Server sollte `AI_PASSIVE`-Flag setzen und sich an alle von `getaddrinfo()` zurückgegeben Adressen binden

Adress- und Namensauflösungen

IPv4	IPv4/IPv6
`gethostbyname()`	`getaddrinfo()`
`gethostbyaddr()`	`getnameinfo()`

- Einfluss auf Rückgabewerte von `getaddrinfo()` und `getnameinfo()` über Funktions-Parameter

→ immer und überall die protokollunabhängigen Funktionen `getaddrinfo()` und `getnameinfo()` verwenden

→ Client sollte jede von `getaddrinfo()` zurückgegebene Adresse versuchen, um zu verbinden

→ Server sollte `AI_PASSIVE`-Flag setzen und sich an alle von `getaddrinfo()` zurückgegeben Adressen binden

Adress-Umwandlungen String/Binary

	IPv4	IPv4/IPv6
String → Binary	`inet_aton()` und `inet_addr()`	`inet_pton()`
Binary → String	`inet_ntoa()`	`inet_ntop()`

- `inet_pton()` und `inet_ntop()` sind reentrant
- `inet_pton()` und `inet_ntop()` benötigen Info über `AF_FAMILY`

→ besser protokollunabhängigere `getaddrinfo()` und `getnameinfo()` verwenden

Adress-Umwandlungen String/Binary

	IPv4	**IPv4/IPv6**
String → Binary	`inet_aton()` und `inet_addr()`	`inet_pton()`
Binary → String	`inet_ntoa()`	`inet_ntop()`

- `inet_pton()` und `inet_ntop()` sind reentrant
- `inet_pton()` und `inet_ntop()` benötigen Info über `AF_FAMILY`

→ besser protokollunabhängigere `getaddrinfo()` und `getnameinfo()` verwenden

Socket API Core Funktionen

Der `socket()` Systemcall ist unverändert:

```
int s;

s = socket(AF_INET, SOCK_STREAM, 0);   /* IPv4 */
s = socket(AF_INET6, SOCK_STREAM, 0); /* IPv6 */
```

Für alle anderen Socket Aufrufe gilt:

→ keine Änderung der Syntax, lediglich Adress-Struktur von `struct sockaddr_storage` auf generisches `struct sockaddr` casten und Adresslängen-Parameter mitgeben

IPv6-Socket-Optionen

Verschiedene IPv6-Socket-Optionen (`IPV6_*`) über `setsockopt()` bzw. `getsockopt()` verfügbar:

- `IPV6_UNICAST_HOPS` zum Ändern des Hop-Limits (entspricht IPv4 `IP_TTL`)
- `IPV6_CHECKSUM` zum Berechnen und Prüfen von Checksummen über alle Pakete auf RAW-Sockets
- verschiedene Multicast-Optionen:
  - `IPV6_JOIN_GROUP` (entspricht IPv4 `IP_ADD_MEMBERSHIP`)
  - `IPV6_LEAVE_GROUP` (entspricht IPv4 `IP_DROP_MEMBERSHIP`)
  - `IPV6_MULTICAST_*` (analog zu IPv4 `IP_MULTICAST_*`)
  - ...
- ...

Adress-Macros

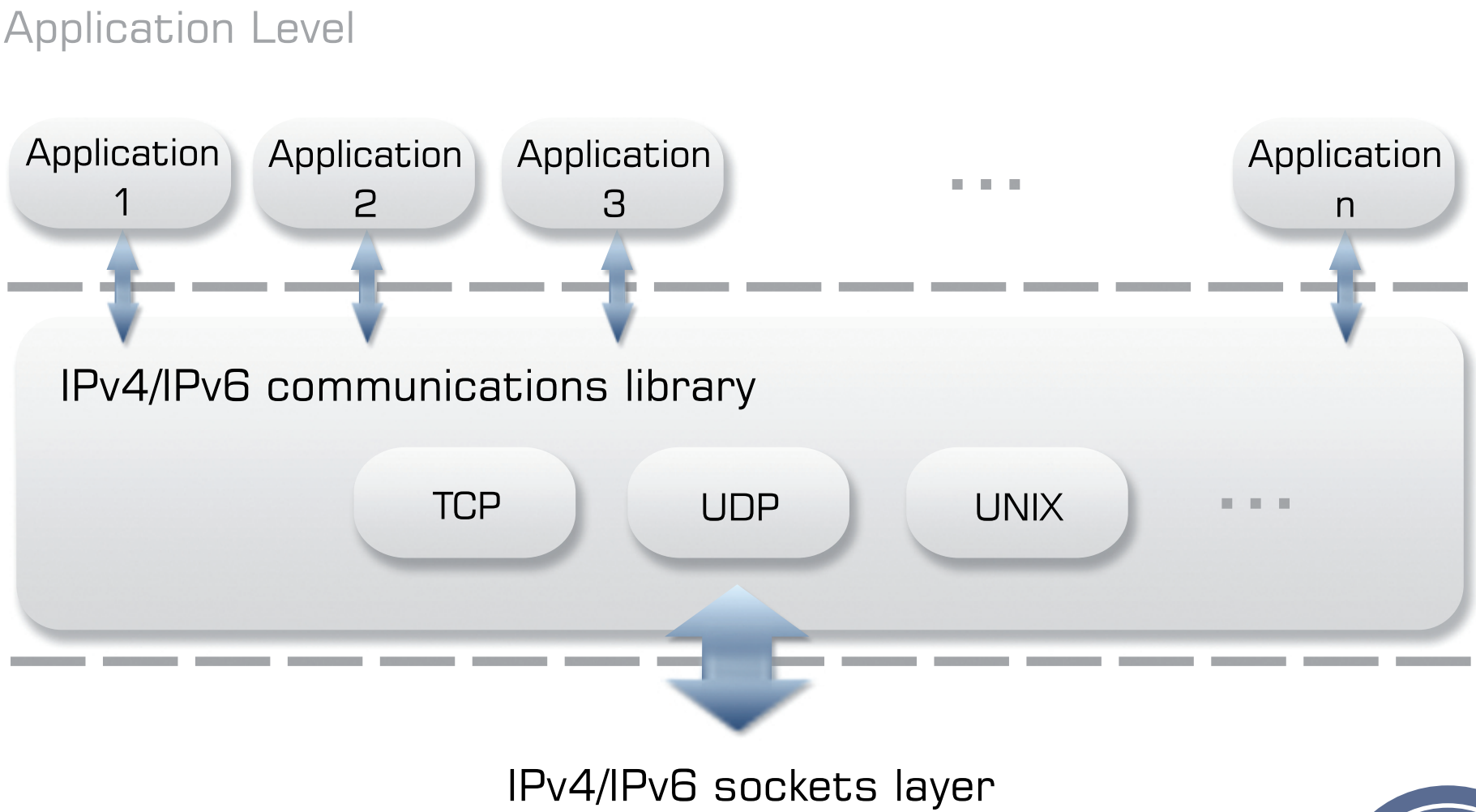
Verschiedene nützliche Konstanten und Macros in `<netinet/in.h>` verfügbar:

IPv4	IPv4/IPv6
`INADDR_ANY`	`const struct in6_addr in6addr_any;`
`INADDR_LOOPBACK`	`const struct in6_addr in6addr_loopback;`

- `IN6_IS_ADDR_LOOPBACK(a)`
- `IN6_ARE_ADDR_EQUAL(a,b)`
- verschiedene Multicast-Macros
- ...

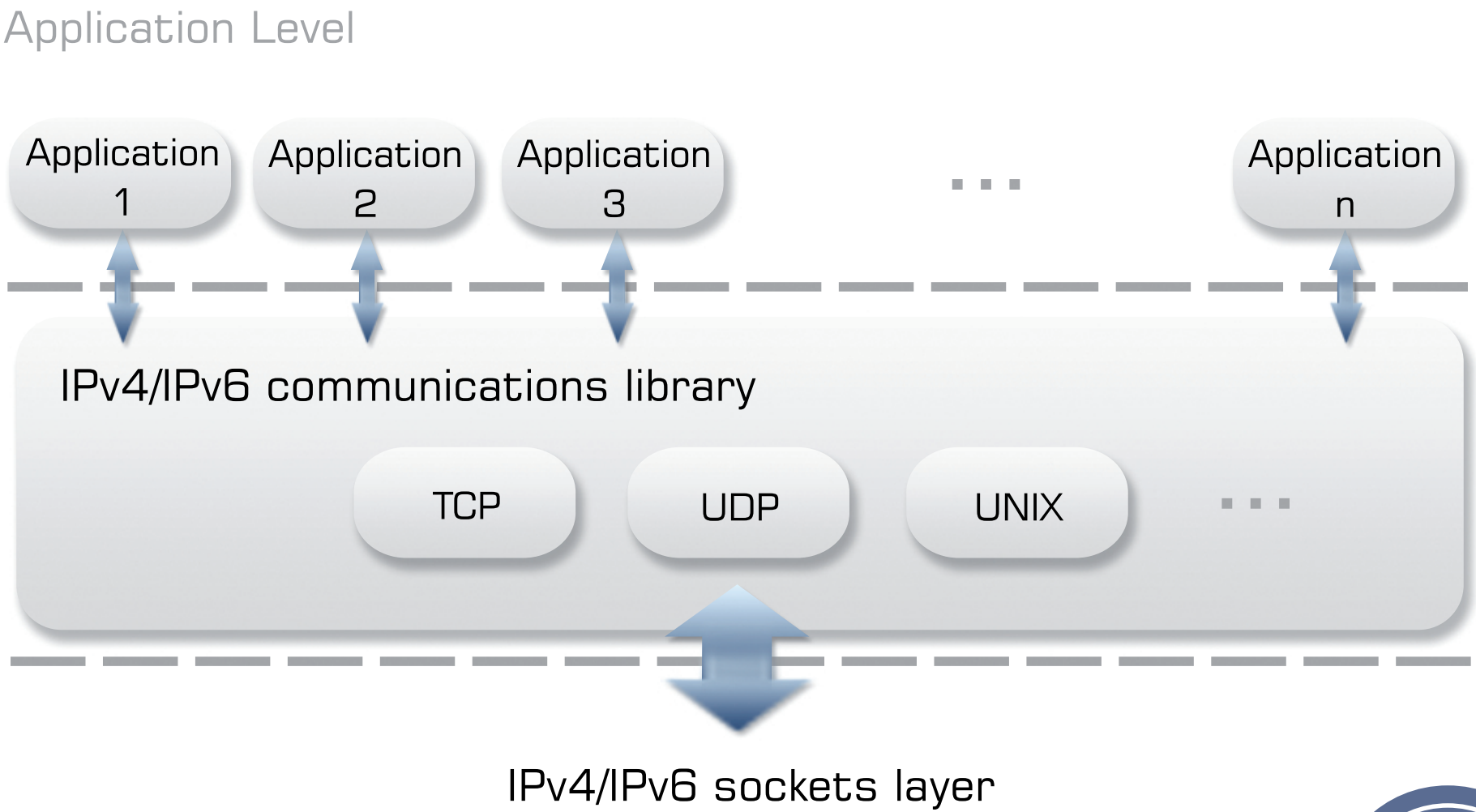
Gliederung

Generelles Anwendungs-Design



→ einfachere Integration neuer Protokolle und Netzwerke

Generelles Anwendungs-Design



→ einfachere Integration neuer Protokolle und Netzwerke

Generelle Design-Empfehlungen

- Schnittstelle zum Transport- und Netzwerk-Modul sollte klein gehalten werden mit einigen wenigen Funktionen
- **Vermeiden** von protokollspezifischem Code (`case AF_INET6`)
- Vermeiden von `#ifdef` und stattdessen Separieren von Code in einzelne Dateien
- Optionen für Experten-Benutzer, um explizit IPv4 oder IPv6 zu benutzen ("`-4`" und "`-6`" als Kommandozeilen-Option oder Parameter in Konfigurationsdatei)

User-Interface-Design Empfehlungen

- Anpassen von Eingabe-/Ausgabe-Interfaces (dynamische Masken-Größen), um längere IPv6-Adressen entgegenzunehmen bzw. darzustellen
- Rückgabe von mehreren Adressen bei Adress-Auflösung sollte im UI-Design berücksichtigt werden
- Prüfen auf Buffer-Overflows resultierend aus längeren Adressen (auch Log-Nachrichten werden länger)

Zusammenfassung

- IPv4/IPv6-fähige Stacks und Anwendungen
- protokollunabhängige Strukturen (`struct sockaddr_storage()`)
- protokollunabhängige Funktionen (`getaddrinfo()/getnameinfo()`)
- Vermeiden von protokollspezifischen (Kontroll-)Strukturen und Funktionen (`inet_ntop()/inet_pton()`)
- Schleifen zum Behandeln mehrerer Adressen bei Address- und Namensauflösung auf Client- und Serverseite
- Nutzen vorhandener Standard Funktionen und Macros zum “Verwalten” von Adressen (vergleichen, umwandeln)

→ IPv6-Aktivitäten an der Universität Potsdam:
`http://www.ipv6-showcase.de/`
`http://www.ipv6-ids.de/`

Marc Blanchet, André Cormier, and Florent Parent.

Porting applications to IPv6: simple and easy!, 2000.

Available from: http://www.viagenie.qc.ca/en/ipv6/presentations/IPv6%20porting%20appl_v1.pdf.

Eva M. Castro.

Porting applications to IPv6 HowTo, 2008.

Available from: http://gsyc.escet.urjc.es/~eva/IPv6-web/ipv6.html.

Michael J. Donahoo and Kenneth L. Calvert.

TCP/IP Sockets in C. Practical Guide For Programmers.

Morgan Kaufmann, second edition, 2009.

Jun ichiro Itoh.

Implementing AF-independent application, 2003.

Available from: http://www.kame.net/newsletter/19980604/.

Ken Renard.

IPv6 for Developers, 2005.

Available from: www.wareonearth.com/whitepapers/IPv6Programming.pps.

Richard W. Stevens, Bill Fenner, and Andrew M. Rudoff.

Unix Network Programming, Volume 1: The Sockets Networking API.

Addison-Wesley Professional Computing Series, third edition, 2003.

You TaeWan.

IPv6 Network Programming, 2004.

Available from: http://www.ipv6.or.kr/summit2004/proceeding/3.%20Tutorial%20II/TII-2.pdf.

Mauro Tortonesi.

Best practices in IPv6-enabled networking software development, 2003.

Available from: ftp://ftp.deepspace6.net/pub/ds6/presentations/en/linuxkongress2003.pdf.