Contiki Tutorial

Sven Zehl, Thomas Scheffler

Beuth Hochschule für Technik Berlin

24. September 2013

Contiki

The Open Source OS for the Internet of Things

$\Rightarrow$ Contiki ist ein Open Source Betriebssystem für das Internet of Things. Contiki verbindet kleine, günstige, ressourcenarme Geräte mit dem Internet.

Contiki

The Open Source OS for the Internet of Things

- Das Internet of Things (Internet der Dinge), beinhaltet alle Technologien, welche Geräte jeglicher Art und Grösse mit dem Internet verbinden (Kühlschränke, Industrieroboter, Glühbirnen, Spielzeug, Steckdosen...).
- Das Internet of Things ermöglicht somit, dass Software den Zugang zur physischen Welt erlangt.

Was bietet Contiki?

- Multitasking mit Protothreads (globaler Stack)
- TCP/IP mit IPv6 Support über μIP
- Arbeitet auf 8 Bit Mikrocontrollern
- benötigt nur 10kByte RAM
- Programmierung in C

Protothreads und Contiki Prozesse

Protothreads

- kooperatives Multitasking
- Kontextwechsel nur an speziell vorgesehenen Programmstellen möglich
- implementiert durch `switch/case`-Routinen (deshalb besser nicht in den eigenen Programmen verwenden!)
- kein eigener Stapelspeicher
- lokale Variablen müssen statisch oder global definiert werden, wenn sie über einen Kontextwechsel hinweg erhalten bleiben sollen!

Contiki Prozesse I

- basierend auf Protothreads
- wie Protothreads basierend auf switch Anweisungen, aktivierbar durch Präprozessormakros
- dem Betriebssystem mitgeteilt über `PROCESS([prozessname], "[Prozessbeschreibung]");`
- Autostarteintrag über Eintrag in `AUTOSTART_PROCESSES(&[prozessname]);`

Contiki Prozesse II

- Prozessbeginn durch Makro `PROCESS_BEGIN()` (bindet Prozess in globale switch Routine ein)
- Ende des Prozesses muss mit `PROCESS_END()` markiert werden (beendet switch)
- Der Prozess läuft solange weiter bis ein *blocking macro* erreicht wird oder der Prozess vorzeitig beendet wird durch `PROCESS_EXIT()`.

Contiki Prozesse III - *blocking macros*

- `PROCESS_WAIT_EVENT()`: unterbreche und warte bis eine Event-Nachricht an den Prozess gepostet wird.
- `PROCESS_WAIT_EVENT_UNTIL(cond)`: unterbreche und warte bis eine Event-Nachricht an den Prozess gepostet wird und eine bestimmte Bedingung eingetroffen ist.
- `PROCESS_WAIT_UNTIL(cond)`: warte bis die Bedingung eingetroffen ist. Wenn die Bedingung beim Aufruf schon wahr ist, wird der Prozess nicht unterbrochen.
- `PROCESS_WAIT_WHILE(cond)`: warte solange die Bedingung wahr ist. Wenn die Bedingung beim Aufruf nicht erfüllt ist, wird der Prozess nicht unterbrochen.

Contiki Timer

Contiki Timer

- Contiki enthält aktive und passive Timer Strukturen:
  - `struct timer`: passiver Timer, enthält die Zeit seit seinem Start, Prozesse müssen den Ablauf selbst überprüfen.
  - `struct stimer`: passiver Timer, ähnlich zu `struct timer`, zählt jedoch in Sekunden, dadurch längere Zeitmessung möglich.
  - `struct etimer`: aktiver Timer, sendet bei Ablauf eine Event-Nachricht and den Prozess welcher ihn gestartet hat.
  - `struct ctimer`: aktiver Timer, ruft bei Ablauf eine Funktion auf.
  - `struct rtimer`: Echtzeit Timer, ruft exakt bei Ablauf eine Funktion auf.
- Jede Timer Bibliothek besitzt eine spezielle API zur Steuerung,
  $\Rightarrow$Details: https://github.com/contiki-os/contiki/wiki/Timers

Protothreads - Beispiel

```
1  #include "contiki.h"
2  #include <stdio.h>
3
4  PROCESS(pt_example, "Protothread Example Process");
5
6  AUTOSTART_PROCESSES(&pt_example);
7
8  PROCESS_THREAD(pt_example, ev, data)
9   {
10    /*Timer Variable statisch deklariert, damit sie nach dem
          Kontextwechsel noch verfügbar ist*/
11    static struct etimer et;
12
13    PROCESS_BEGIN();
14
15    etimer_set(&et, (CLOCK_SECOND*20));
16
17    while(1) {
18      /*Warte nun bis die 20s abgelaufen sind, solange können
            andere Protothreads den Prozessor nutzen*/
19      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
20
21      etimer_reset(&et);
22
23      printf("Timer abgelaufen, starte Timer neu...\n");
24    }
25    PROCESS_END();
26  }
```

μIP

μIP I

- micro Internet Protocol Stapel, speziell für ressourcenarme Geräte
- RFC-Standard kompatibel
- Unterstützung von IP, ICMP, UDP und TCP
- Schnittstelle für TCP über ProtoSockets (BSD-Socket ähnlich, realisiert über Protothreads)
- Schnittstelle für UDP über die μIP raw API
- Außerdem:

  $\Rightarrow$ einzelner statischer Paketbuffer
  $\Rightarrow$ Prozesse müssen ihre Daten abholen bevor neue Pakete empfangen werden können.

μIP II - UDP Kommunikation 1

- Eine neue UDP Verbindung wird mit der Funktion `udp_new(const uip_ipaddr_t *ripaddr, uint16_t port, void *appstate)` aufgebaut, welche einen Pointer auf die unten gezeigte Struktur zurückliefert.
- Diese Struktur wird verwendet um die neue UDP Verbindung zu verwalten.

```
1  struct uip_udp_conn {
2    /**< IP Adresse des Remote Peers. */
3    uip_ipaddr_t ripaddr;
4    /** Lokale Port Nummer in Network Byte Order */
5    uint16_t lport;
6    /**< Remote Port Nummer in Network Byte Order */
7    uint16_t rport;
8    /**< Default Time-To-Live*/
9    uint8_t  ttl;
10   /** Applikationspezifische Struktur, ermöglicht der
         Applikation das Speichern von Daten*/
11   uip_udp_appstate_t appstate;
12 };
```

μIP II - UDP Kommunikation 2 - Beispiel UDP Server

```
1  static struct uip_udp_conn *udpconn;
2
3  PROCESS_THREAD(example_udp_server_process, ev, data)
4  {
5    PROCESS_BEGIN();
6    /*Starte neue UDP Verbindung mit IP 0.0.0.0 und Port 0, */
7    /* d.h. akzeptiere jede ankommende Verbindung*/
8    udpconn = udp_new(NULL, UIP_HTONS(0), NULL);
9    /*Setze den Port auf dem gelauscht wird auf 50000*/
10   /*HTONS() übersetzt zu Network Byte Order*/
11   udp_bind(udpconn, UIP_HTONS(50000));
12   printf("listening on UDP port %u\n", UIP_HTONS(udpconn->
         lport));
13   while(1) {
14     /* Warte bis ein TCP/IP event eintrifft */
15     PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
16     /*Rufe die Handler-Funktion auf*/
17     bsp_udphandler();
18   }
19   PROCESS_END();
20 }
```

$\Rightarrow$ Die Handler-Funktion muss vom Anwendungs-Programmierer festgelegt werden und beinhaltet die Weiterverarbeitung mit den Daten.

μIP II - UDP Kommunikation 2 - Beispiel UDP Server

```
1  void bsp_udphandler(void)
2  {
3    char buf[MAX_PAYLOAD_LEN];
4    if(uip_newdata())
5    {
6      /*Zeige Benutzer den Inhalt der empfangenen Nachricht*/
7      /*Setze letztes Byte zu Null, für String Ende*/
8      ((char *)uip_appdata)[uip_datalen()] = 0;
9      printf("Server received: '%s'", (char *)uip_appdata);
10     /*Verwende die Quell- als Zieladresse für Antwort */
11     uip_ipaddr_copy(&udpconn->ripaddr, &UDP_IP_BUF->
           srcipaddr);
12     udpconn->rport = UDP_IP_BUF->srcport;
13     /*Schreibe Antwort Daten in Buffer*/
14     sprintf(buf, "Hello from the server! (%d)");
15     /*Versende das Antwort Packet*/
16     uip_udp_packet_send(udpconn, buf, strlen(buf));
17     /*Setze Adresse/Port in Verbindungsstruktur auf Null,*/
18     /*um von jedem Absender Daten empfangen zu können*/
19     memset(&udpconn->ripaddr, 0, sizeof(udpconn->ripaddr));
20     udpconn->rport = 0;
21   }
22 }
```

μIP II - TCP Kommunikation 1

- Die Funktion `void psock_init(struct psock *psock, uint8_t *buffer, unsigned int buffersize)` wird verwendet um ein neues Protosocket zu initialisieren.
- Es wird eine Struktur vom Typ `struct psock` beschrieben, in welcher der Protothread verwaltet wird.

```
1  struct psock {
2    struct pt pt, psockpt;
3    /*Zeiger auf die Daten die zu Senden sind*/
4    const uint8_t *sendptr;
5    /*Zeiger auf die Daten die zu Lesen sind*/
6    uint8_t *readptr;
7    /* Zeiger auf den Input Buffer*/
8    uint8_t *bufptr;
9    /*Anzahl an Bytes welche noch zu Senden sind*/
10   uint16_t sendlen;
11   /*Anzahl an Bytes welche noch zu Lesen sind*/
12   uint16_t readlen;
13   struct psock_buf buf; /*Input Buffer Struktur*/
14   unsigned int bufsize; /*Größe Input Buffer*/
15   unsigned char state;  /*Protosocket State*/
16 };
```

μIP II - TCP Kommunikation 2 - Beispiel TCP Server

```
1  static struct psock ps; //Protothread Struktur
2  PROCESS_THREAD(example_psock_server_process, ev, data)
3  {
4   PROCESS_BEGIN();
5   /*Festlegung des Listening Ports*/
6   tcp_listen(UIP_HTONS(1010));
7   while(1)
8   {
9    printf("TCP Server is running\n");
10   /*Warte bis ein Packet eingetroffen ist*/
11   PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
12    /*Bei Verbindungsanfrage initialisiere Protosocket*/
13    if(uip_connected())
14    {
15     PSOCK_INIT(&ps, buffer, sizeof(buffer));
16     while(!(uip_aborted()||uip_closed()||uip_timedout()))
17     {
18      PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
19       handle_connection(&ps);
20     }/*Aufruf der Handler-Funktion*/
21    }
22  }
23  PROCESS_END();
24 }
```

$\Rightarrow$ Die Handler-Funktion wird vom Anwendungs-Programmierer festgelegt.

μIP II - TCP Kommunikation 2 - Beispiel TCP Server

```
1  static char buffer[100]; //Nachrichtenbuffer
2
3  static PT_THREAD(handle_connection(struct psock *p))
4  {
5    /*Protosocket Protothread Start*/
6    PSOCK_BEGIN(p);
7
8    /*Sende Willkommensnachricht*/
9    PSOCK_SEND_STR(p, "Welcome, your request will be processed
         ...\n");
10
11   /*Warte auf Nachricht von Client, \n zeigt Datenende an*/
12   PSOCK_READTO(p, '\n');
13   printf("TCP Server received: %s \n",buffer);
14
15   /*Sende Antwort an Client*/
16   PSOCK_SEND_STR(p, "You wrote: ");
17   PSOCK_SEND(p, buffer, PSOCK_DATALEN(p));
18   PSOCK_SEND_STR(p, "Good bye!\r\n");
19
20   /*Schließe Protosocket*/
21   PSOCK_CLOSE(p);
22   /*Beende Protosocket Protothread*/
23   PSOCK_END(p);
24 }
```

Anwendungsentwicklung unter Contiki OS

Anwendungsentwicklung Contiki OS

- Entwicklungsumgebung Instant Contiki
- VMware Image basierend auf Ubuntu
- enthält alle benötigten Programme, Compiler, Toolchain und Simulatoren.
  $\rightarrow$ Alternativ kann auch eine Cygwin Umgebung auf Windows eingerichtet werden, jedoch müssen dann alle zusätzliche Softwarepackete manuell installiert werden.
- Kompilierung mithilfe von `make`-Files

Anwendungsentwicklung Contiki OS - make I

- Tool der Softwareentwicklung
- für Projekte mit vielen Quellcodedateien
- führt alle benötigten Arbeitsschritte zum Erstellen der ausführbaren Datei durch (übersetzen, linken, kopieren, umbenennen usw...)
- die Anweisungen für `make` werden immer in einem `Makefile` festgelegt.

Anwendungsentwicklung Contiki OS - make II

- Contiki hat mehrere Makefiles, mit unterschiedlichen Aufgaben
- `Makefile.include`, befindet sich im obersten Ordner des Contiki Quellcodes, enhält alle Anweisungen um Contiki OS zu kompilieren.
- `Makefile.[TARGET]`, wobei für TARGET die verwendete Hardware eingtragen ist, z.B. `avr-zigbit` (Makefile.avr-zigbit), befindet sich im Ordner `platform/avr-zigbit`. Enthält alle hardware spezifischen Anweisungen um Contiki OS für eine bestimmte Zielplattform zu kompilieren.

Anwendungsentwicklung Contiki OS - make III

- `Makefile.[APP]`, wobei APP dem Namen der Applikation entspricht, welche in Contiki eingebunden werden soll. Befindet sich im Ordner /app/[Anwendung].
- `Makefile` ohne Dateierweiterung. Es ist das eigentliche Makefile und befindet sich im Projekt oder examples Ordner. Die Kompilierung startet ab diesem Punkt durch das Kommando `make`. Verknüpfungen zu anderen Makefiles werden hier gespeichert. Außerdem ist es möglich Schritte die nach der Kompilierung folgen sollen hier ebenfalls einzutragen (z.B. avr-size).

Anwendungsentwicklung Contiki OS - make IV - Beispiel

⇒ Beispiel Projekt `contiki_tut`, keine einzubindende APP, Projekt ist gleichzeitig die Applikation. Beide Makefiles befinden sich im gleichen Ordner z.B. `examples/contiki_tut/`.

Makefile | Makefile.contiki_tut

```
1  all:
2      ${MAKE} TARGET=avr-zigbit -f Makefile.contiki_tut contiki_tut.elf
3
```

legt Zielplatform fest

verweist auf Makefile.contiki_tut

Ausgabedatei

Makefile | Makefile.contiki_tut

```
1  CONTIKI_PROJECT = contiki_tut
2  all: $(CONTIKI_PROJECT)
3
4  CONTIKI = ../..
5  UIP_CONF_IPV6=1
6  include $(CONTIKI)/Makefile.include
```

Projektname

Pfad zum Contiki Wurzelordner

Aktivierung von IPv6

Verweis auf Contiki Makefile.include

Anwendungsentwicklung Contiki OS - Contiki Ordnerstruktur I

Der Contiki Quellcode ist in eine bestimmte Ordnerstruktur gegliedert

- **Contiki Wurzelordner** → oberster Ordner, hier befindet sich z.B. das globale Makefile.include
  - **apps** → Hier befinden sich alle Contiki Applikationen, der Quellcode einzelnen Apps kann über das Makefile mit dem Befehl APPS=[name des App-Ordners] hinzugefügt werden
  - **core** → In diesem Ordner befindet sich das eigentliche Betriebssystem, u.a auch der Ordner `net`, wo der komplette Quellcode für die Netzwerkkommunikation zu finden ist. Hier ist auch der Ordner `sys`, welcher den Quellcode für das Prozessmanagement und die Timer enthält.
  - **cpu** → Hier findet sich der Quellcode für die unterstützenden Prozessoren, also z.B. den Ordner `avr` für die ATmega Prozessoren oder der Ordner `arm` für ARM Prozessoren. In den jeweiligen Unterordnern sind auch die Treiber für passende Funkchips abgelegt.

Anwendungsentwicklung Contiki OS - Contiki Ordnerstruktur II

Der Contiki Quellcode ist in eine bestimmte Ordnerstruktur gegliedert

- **Contiki Wurzelordner**
  - **doc** → Hier ist einiges an Dokumentation sowie Beispielcode für und über Contiki zu finden.
  - **examples** → Dieser Ordner enthält fertige Contiki Projekte, er könnte auch genauso gut `projects` heissen. Er kann auch zum Erstellen von neuen Projekten genutzt werden.
  - **platform** → enthält alle hardwarespezifischen Einstellungen für die verwendete Plattform. Hierzu gleich mehr auf den folgenden Folien.
  - **tools** → Bestimmte Werkzeuge die bei der Arbeit mit Contiki hilfreich sind, z.B. der Simulator `cooja` oder der Quellcode für den SLIP Border Router auf der PC Seite.

Anwendungsentwicklung Contiki OS - der Platform Ordner

Contiki platform Ordner - /platform/[device], z.B. /platform/avr-zigbit/.
$\Rightarrow$ enthält alle hardwarespezifischen Einstellungen für die verwendete Plattform.

- Datei `contiki.conf`, enthält alle Veränderungen am Contiki Quellcode, welche für die spezielle Hardwareplatform benötigt werden. Unter anderem:
  - Einstellungen für μIP, manuelles Ab-/Zuschalten von TCP/UDP und ICMP, Maximas für Neighbors, Verbindungen, ListenPorts uvm...
  - Einstellungen für 6LoWPAN, Headerkompressionsalgorithmus
  - Einstellungen für 802.15.4, Funktreiber, MAC-Layer
  - Einstellungen für RPL Routing, z.B. UIP_CONF_ROUTER, jeder Konten ist gleichzeitig Router, benötigt für vermaschtes Routing.
  - Auswahl der UART für SLIP Nutzung

Anwendungsentwicklung Contiki OS - der Platform Ordner

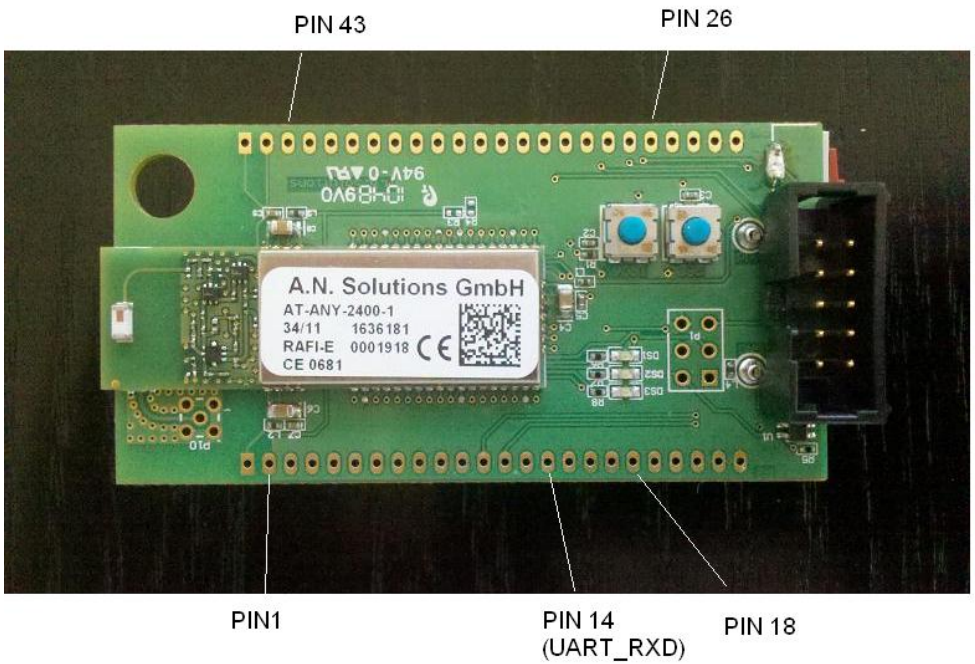
Contiki platform Ordner - /platform/[device], z.B. /platform/avr-zigbit/.
$\Rightarrow$ enthält alle hardwarespezifischen Einstellungen für die verwendete Plattform.

- Datei `contiki-[device].c`, z.B. `contiki-avr-zigbit.c` enthält alle Funktionen zur Low Level Initialisierung, sowie einige Hardwareeinstellungen:
  - Einstellungen für UART, Baudrate, Port, Parität, Auswahl der UART für printf() usw...
  - Einstellung der MAC Adresse (daraus wird auch die IPv6 Adresse generiert)
  - Einstellung des 802.15.4 Funkkanals, sowie PAN-ID
  - Einstellungen für Announce Boot, der printf()-Meldung beim Bootvorgang von Contiki

Hardware

AT-ANY-2400

- AT-ANY-2400 von AN Solutions Dresden
- basierend auf dem ATmega1281 (Microcontroller) und dem AT86RF231 (2,4 GHz IEEE 802.15.4 Funkchip) von Atmel
- ähnlich dem Zigbit Modul von Atmel, deshalb von Contiki unterstützt (Contiki `TARGET=avr-zigbit`)
- Entwicklungsboard AT-ANY-BRICK, mit break-out-board um Zugang zu den Pins des Mikrocontrollers zu bekommen



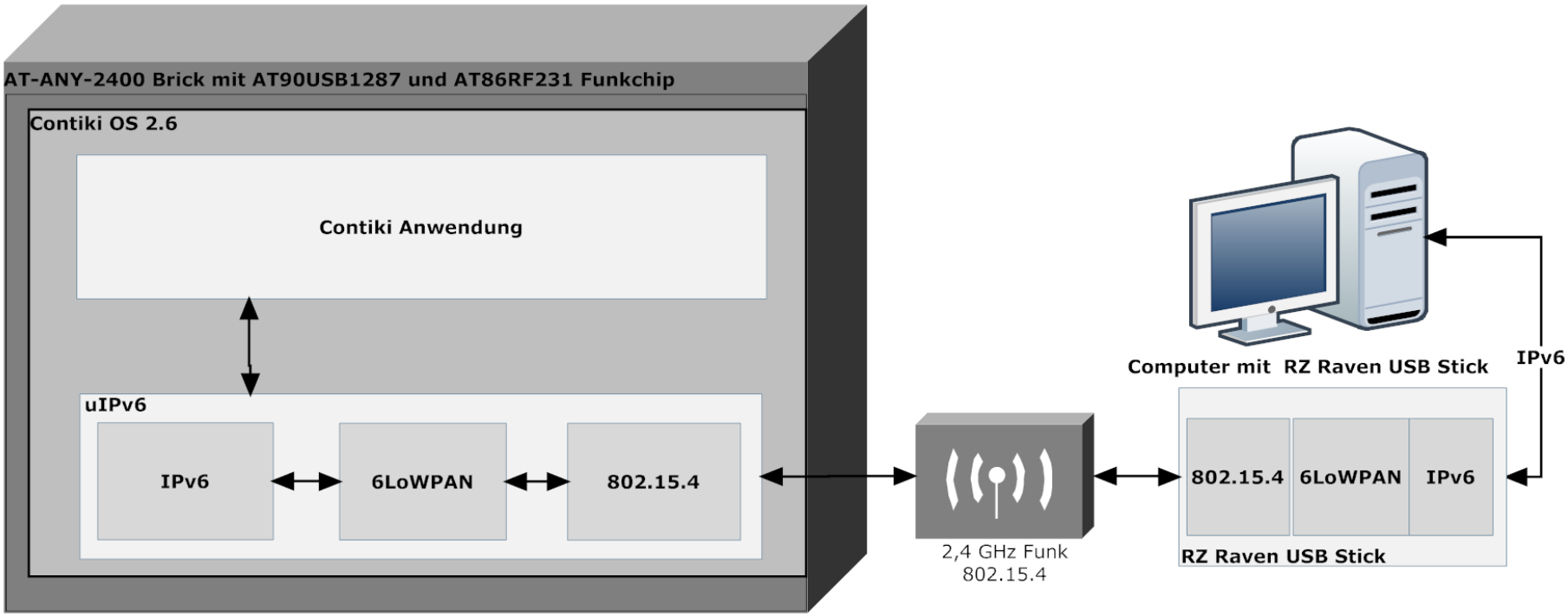
ATmega1281

- Flash (Kbytes): **128**
- SRAM (Kbytes): **8**
- EEPROM (Bytes): **4096**
- Max. Operating Frequency: **16 MHz**
- CPU: **8-bit AVR**
- **SPI, I2C, UART**
- **10 Bit** Analog-Digital-Wandler
- Operating Voltage (Vcc): **1.8 to 5.5**

RZ Raven USB Stick

- RZ Raven USB Stick, 2,4GHz IEEE 802.15.4 Funkschnittstelle für den PC
- basierend auf dem AT90USB1287 (Microcontroller) und dem AT86RF230 (2,4 GHz IEEE 802.15.4 Funkchip) von Atmel
- direkt von Contiki unterstützt (`TARGET=avr-ravenusb`)
- AT90USB1287 Spezifikation:
  - Flash (Kbytes): **128**
  - SRAM (Kbytes): **8**
  - EEPROM (Bytes): **4096**

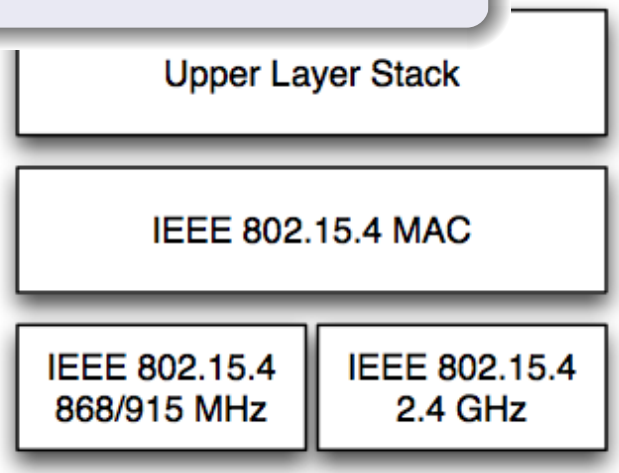
prinzipieller Aufbau



IEEE 802.15.4 - Funklayer

Wichtiger Funkstandard für Home Networks, Industrie- und Gebäudeautomatisierung

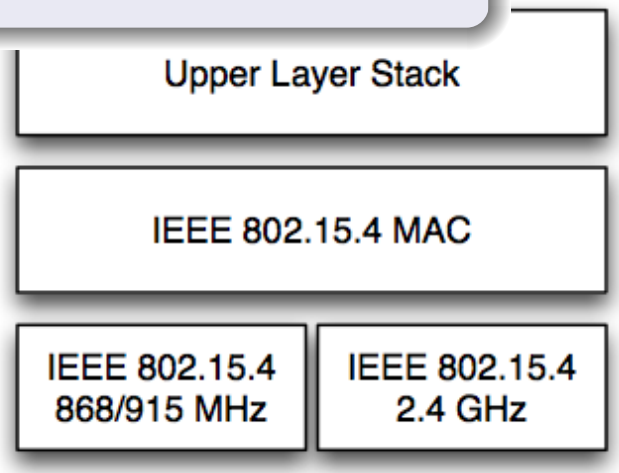
- Unterstützt drei Übertragungsverfahren
  - 20 kbit/s bei 868 MHz
  - 40 kbit/s bei 915 MHz
  - 250 kbit/s bei 2.4 GHz (DSSS)
- Beaconless mode
  - Einfacher CSMA Algorithmus
- Beacon mode mit Superframe
  - Hybrider TDMA-CSMA Algorithmus



IEEE 802.15.4 - Funklayer

Wichtiger Funkstandard für Home Networks, Industrie- und Gebäudeautomatisierung

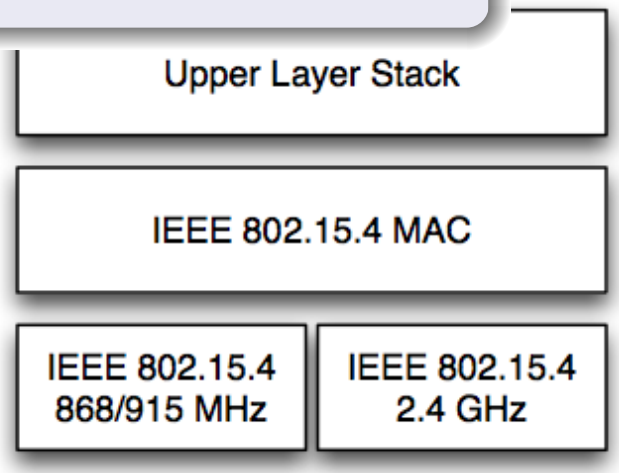
- Unterstützt drei Übertragungsverfahren
  - 20 kbit/s bei 868 MHz
  - 40 kbit/s bei 915 MHz
  - 250 kbit/s bei 2.4 GHz (DSSS)
- Beaconless mode
  - Einfacher CSMA Algorithmus
- Beacon mode mit Superframe
  - Hybrider TDMA-CSMA Algorithmus
- Bis zu 64k Netzknoten mit 16-bit Adressen
- Erweiterungen:
  - IEEE 802.15.4a, 802.15.4e, 802.15.5



IEEE 802.15.4 - Funklayer

Wichtiger Funkstandard für Home Networks, Industrie- und Gebäudeautomatisierung

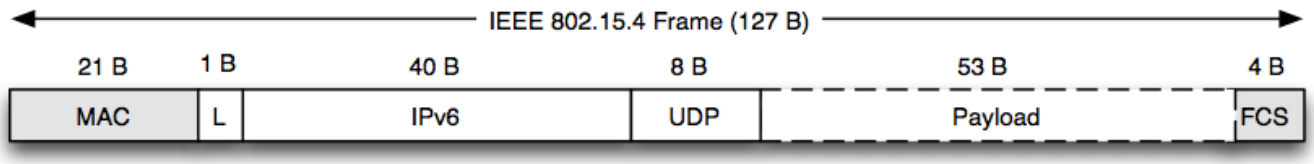
- Unterstützt drei Übertragungsverfahren
  - 20 kbit/s bei 868 MHz
  - 40 kbit/s bei 915 MHz
  - 250 kbit/s bei 2.4 GHz (DSSS)
- Beaconless mode
  - Einfacher CSMA Algorithmus
- Beacon mode mit Superframe
  - Hybrider TDMA-CSMA Algorithmus
- Bis zu 64k Netzknoten mit 16-bit Adressen
- Erweiterungen:
  - IEEE 802.15.4a, 802.15.4e, 802.15.5



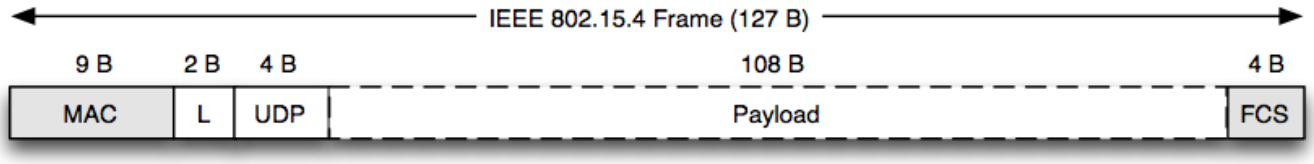
IEEE 802.15.4 - Funklayer

- Framegröße in IEEE 802.15.4 Frames beträgt 127 Byte
- RFC 2460 fordert eine minimale Link-MTU von 1280 Byte

⇒ Um IPv6 Pakete in IEEE 802.15.4 Frames zu übertragen wird eine Anpassungsschicht benötigt: **6LoWPAN**



Full UDP/IPv6 (64-bit addressing)



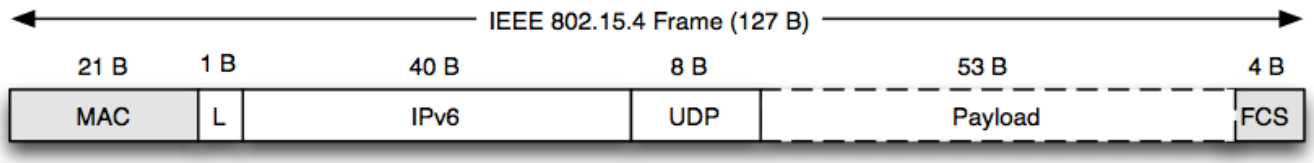
Minimal UDP/6LoWPAN (16-bit addressing)

Quelle: http://6lowpan.net

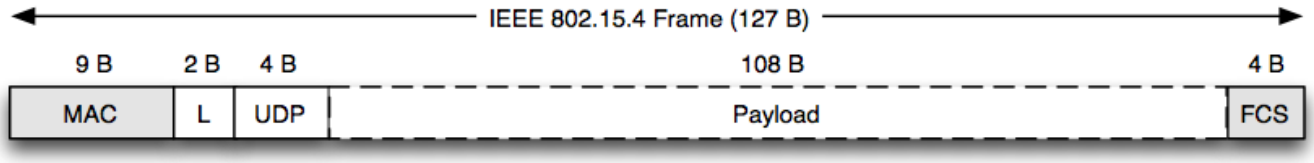
IEEE 802.15.4 - Funklayer

- Framegröße in IEEE 802.15.4 Frames beträgt 127 Byte
- RFC 2460 fordert eine minimale Link-MTU von 1280 Byte

$\Rightarrow$ Um IPv6 Pakete in IEEE 802.15.4 Frames zu übertragen wird eine Anpassungsschicht benötigt: **6LoWPAN**



Full UDP/IPv6 (64-bit addressing)

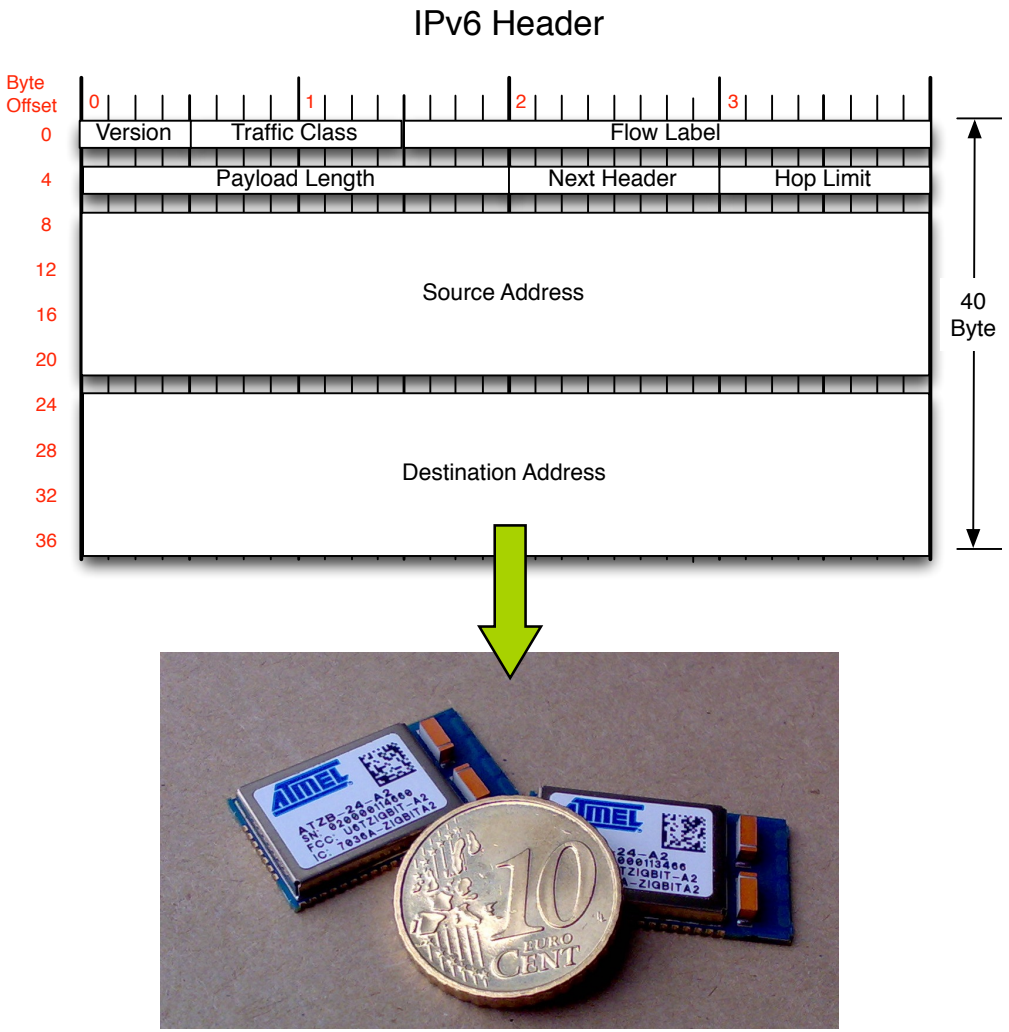


Minimal UDP/6LoWPAN (16-bit addressing)

Quelle: http://6lowpan.net

IPv6 Adaptation Layer:

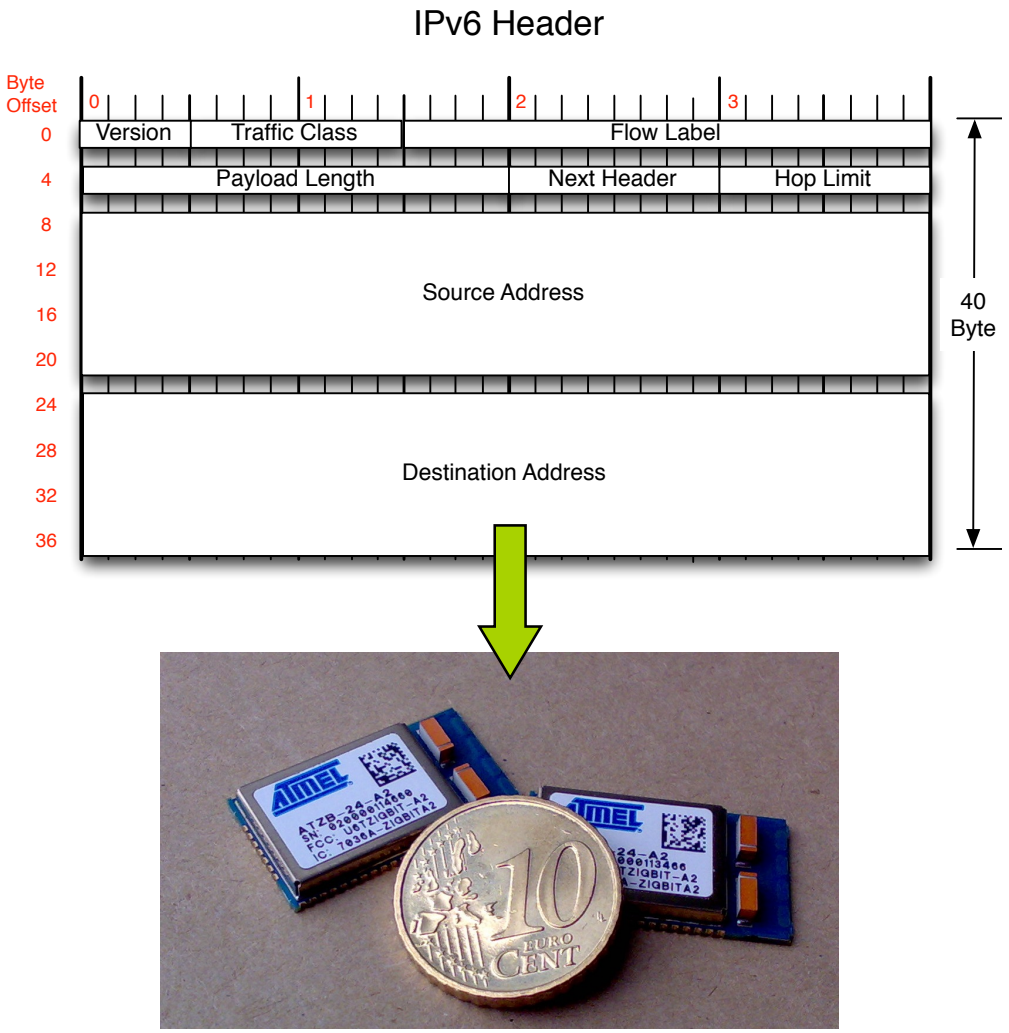
- Standardisiert in RFC 4919, 4944, u.a.
- Stateless Header Compression (IP, UDP)
- Standard Socket API



IPv6 Adaptation Layer:

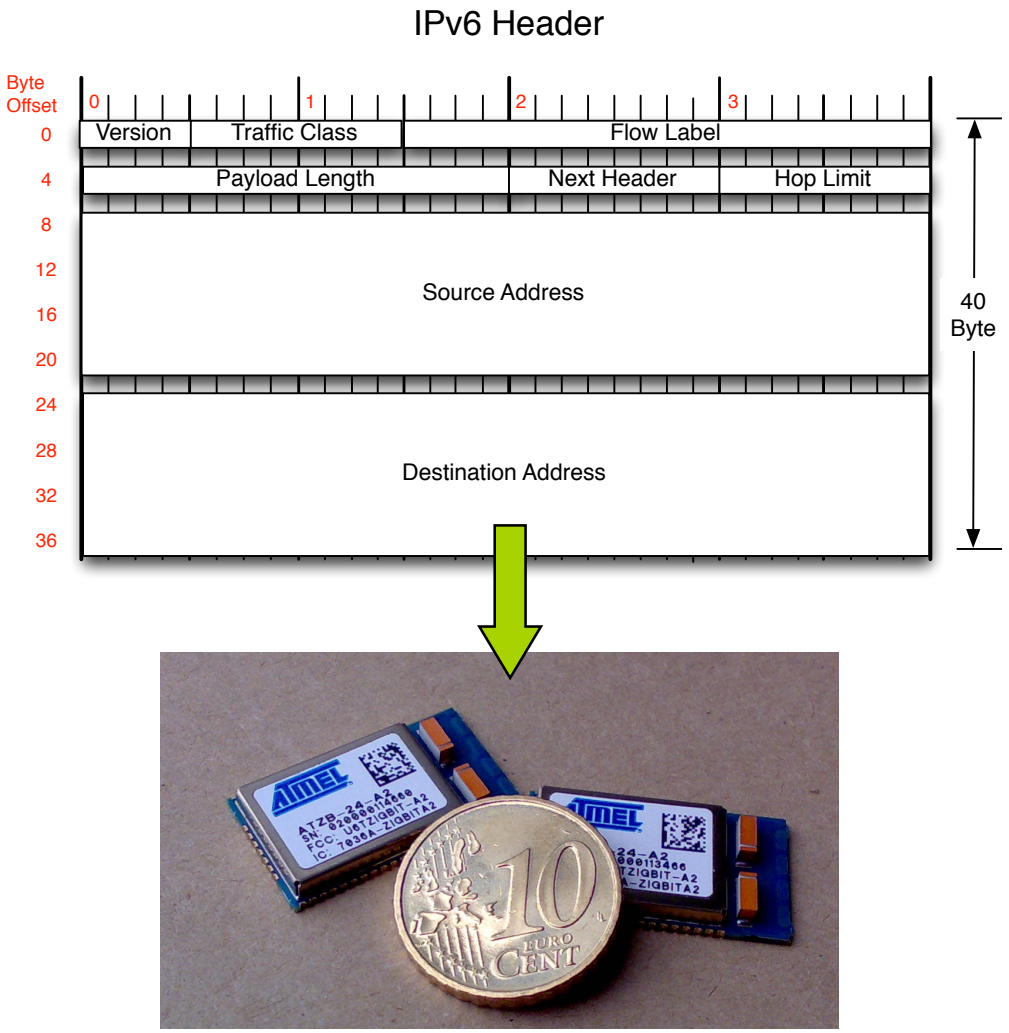
- Standardisiert in RFC 4919, 4944, u.a.
- Stateless Header Compression (IP, UDP)
- Standard Socket API

- Minimaler Code- und Speicherbedarf
- Unterstützung verschiedener Topology Optionen



IPv6 Adaptation Layer:

- Standardisiert in RFC 4919, 4944, u.a.
- Stateless Header Compression (IP, UDP)
- Standard Socket API
- Minimaler Code- und Speicherbedarf
- Unterstützung verschiedener Topology Optionen



6LoWPAN Adressierung

LoWPAN arbeitet nach den folgenden Prinzipien

- Flacher Adressraum (das Funknetz ist ein IPv6 Subnetz)
- Eindeutige MAC Adressen (z.B. 64-bit oder 16-bit)

$\Rightarrow$ Herausforderung für die Hop-by-Hop Paketweiterleitung (Mesh-under, Route-over)

6LoWPAN komprimiert IPv6 Adressen durch:

- Weglassen des IPv6 Prefix
  - Globaler Prefix ist allen Nodes im Netzwerk bekannt
  - Link-local Prefix wird durch das Format der Header-Compression angezeigt

6LoWPAN Adressierung

LoWPAN arbeitet nach den folgenden Prinzipien

- Flacher Adressraum (das Funknetz ist ein IPv6 Subnetz)
- Eindeutige MAC Adressen (z.B. 64-bit oder 16-bit)

$\Rightarrow$ Herausforderung für die Hop-by-Hop Paketweiterleitung (Mesh-under, Route-over)

6LoWPAN komprimiert IPv6 Adressen durch:

- Weglassen des IPv6 Prefix
  - Globaler Prefix ist allen Nodes im Netzwerk bekannt
  - Link-local Prefix wird durch das Format der Header-Compression angezeigt
- Komprimierung der Interface-ID
  - Kann für Link-lokale Nachrichten entfallen
  - Komprimierung für Multihop Dst/Src Adressen

6LoWPAN Adressierung

LoWPAN arbeitet nach den folgenden Prinzipien

- Flacher Adressraum (das Funknetz ist ein IPv6 Subnetz)
- Eindeutige MAC Adressen (z.B. 64-bit oder 16-bit)

$\Rightarrow$ Herausforderung für die Hop-by-Hop Paketweiterleitung (Mesh-under, Route-over)

6LoWPAN komprimiert IPv6 Adressen durch:

- Weglassen des IPv6 Prefix
  - Globaler Prefix ist allen Nodes im Netzwerk bekannt
  - Link-local Prefix wird durch das Format der Header-Compression angezeigt
- Komprimierung der Interface-ID
  - Kann für Link-lokale Nachrichten entfallen
  - Komprimierung für Multihop Dst/Src Adressen
- Multicast Adressen werden komprimiert

6LoWPAN Adressierung

LoWPAN arbeitet nach den folgenden Prinzipien

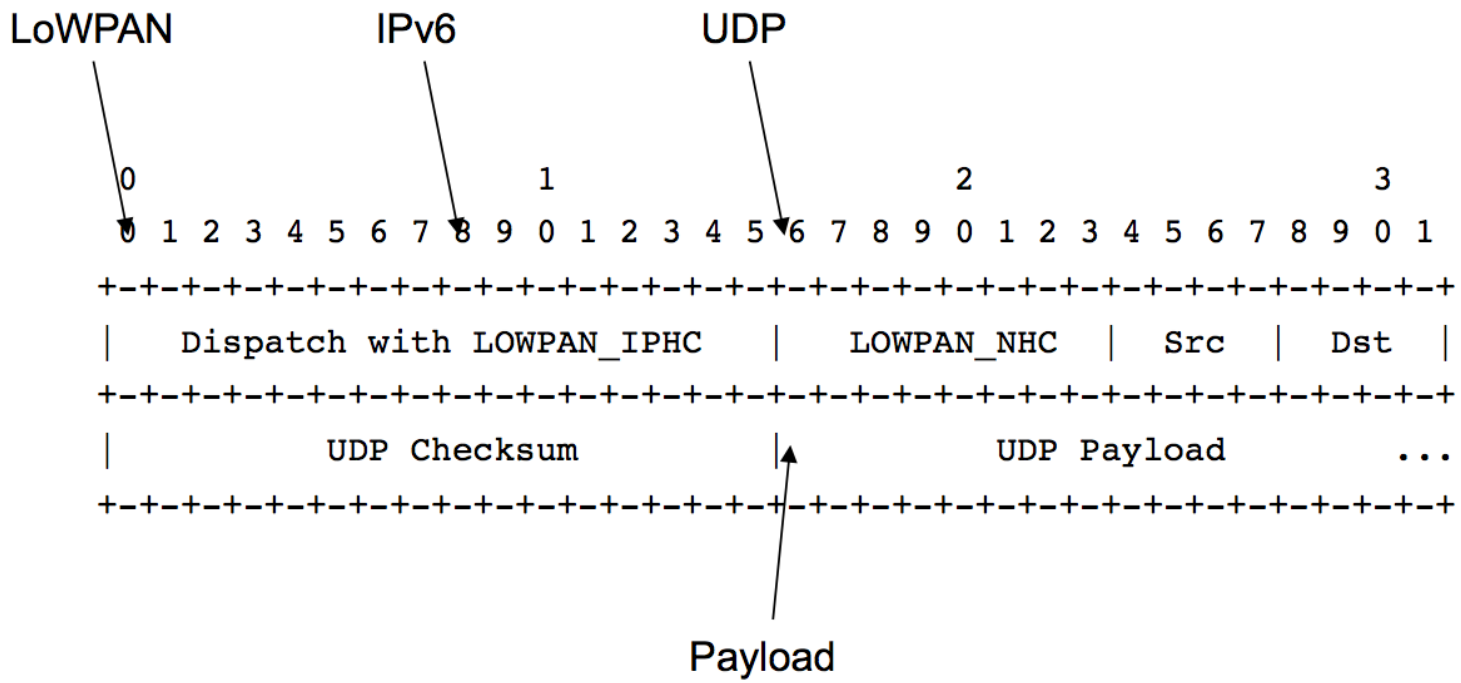
- Flacher Adressraum (das Funknetz ist ein IPv6 Subnetz)
- Eindeutige MAC Adressen (z.B. 64-bit oder 16-bit)

$\Rightarrow$ Herausforderung für die Hop-by-Hop Paketweiterleitung (Mesh-under, Route-over)

6LoWPAN komprimiert IPv6 Adressen durch:

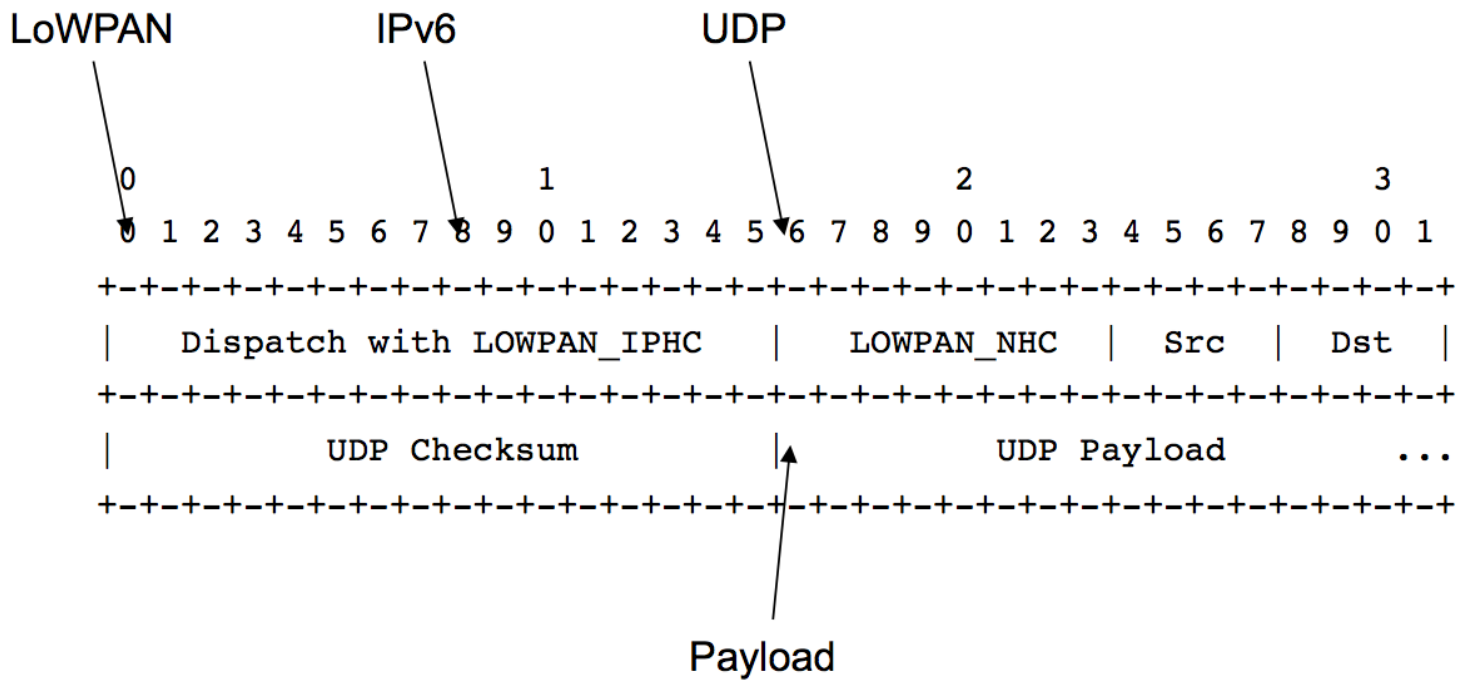
- Weglassen des IPv6 Prefix
  - Globaler Prefix ist allen Nodes im Netzwerk bekannt
  - Link-local Prefix wird durch das Format der Header-Compression angezeigt
- Komprimierung der Interface-ID
  - Kann für Link-lokale Nachrichten entfallen
  - Komprimierung für Multihop Dst/Src Adressen
- Multicast Adressen werden komprimiert

LoWPAN Header Compression



48 → 6 Byte!

LoWPAN Header Compression

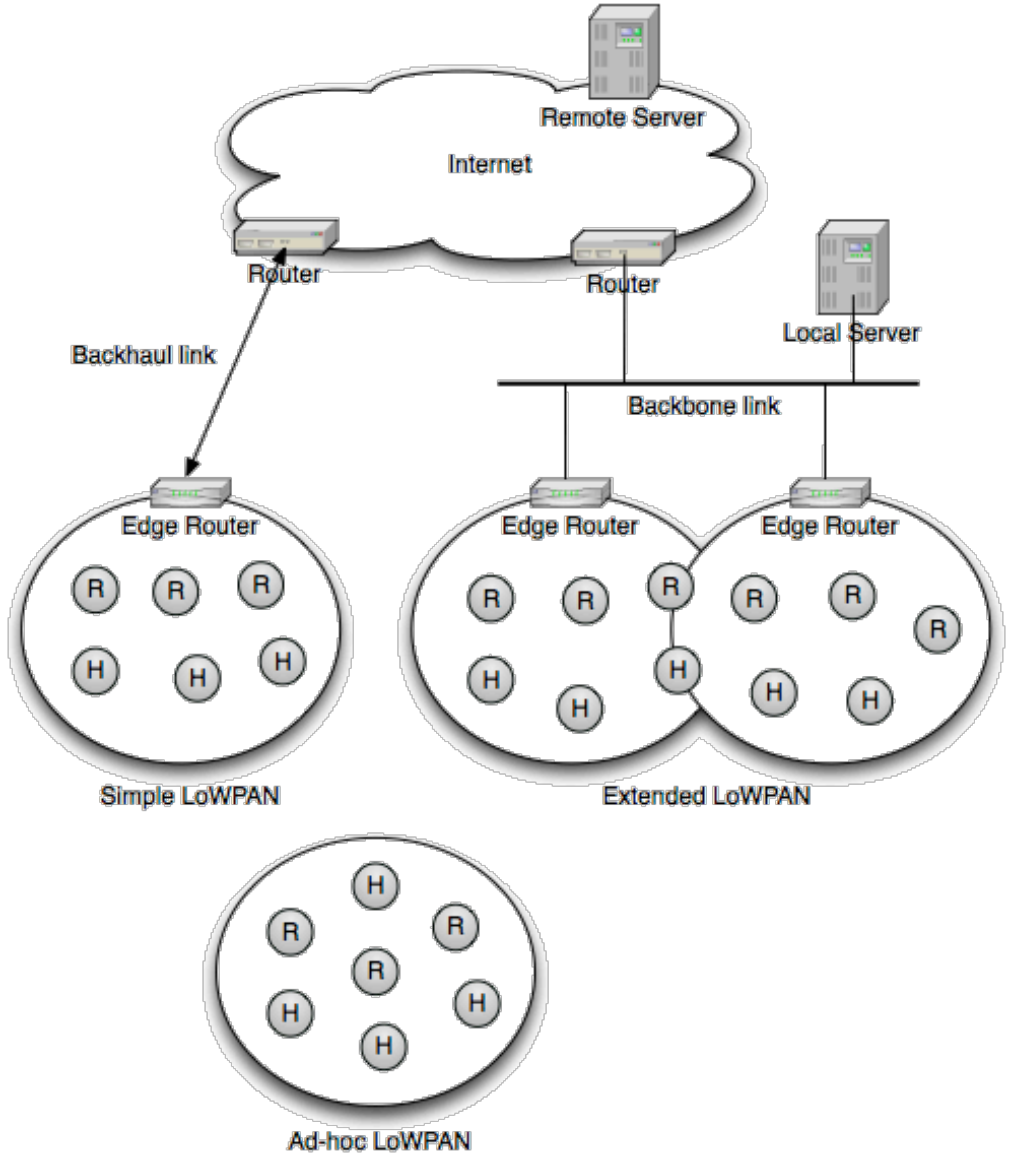


48 → 6 Byte!

LoWPAN Netztopologien

LoWPANs sind Stub-Netzwerke:

- Ad-hoc LoWPAN
  - Keine Routen außerhalb des LoWPAN

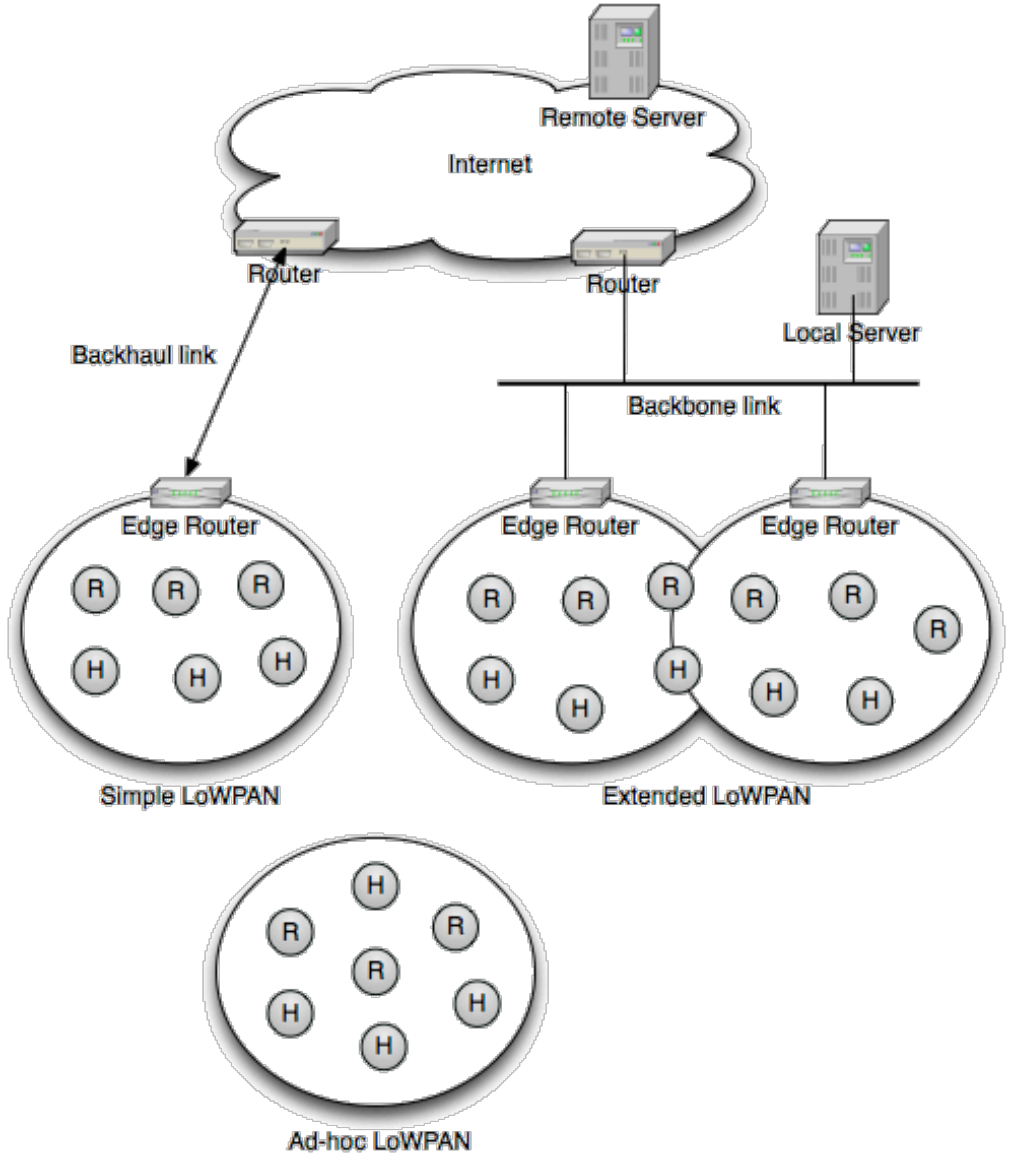


Quelle: http://6lowpan.net

LoWPAN Netztopologien

LoWPANs sind Stub-Netzwerke:

- Ad-hoc LoWPAN
  - Keine Routen außerhalb des LoWPAN
- Simple LoWPAN
  - Single Edge Router

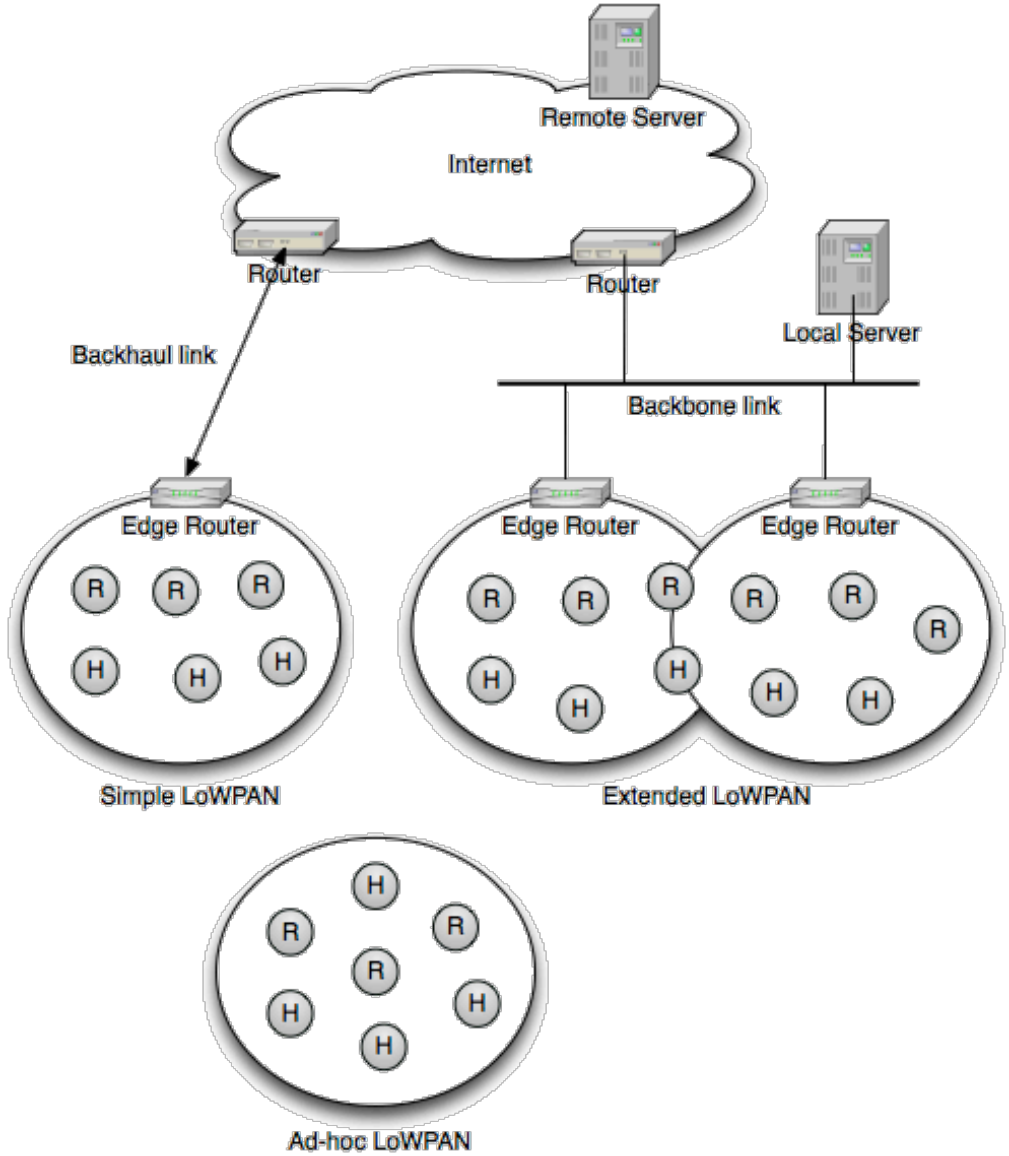


Quelle: http://6lowpan.net

LoWPAN Netztopologien

LoWPANs sind Stub-Netzwerke:

- Ad-hoc LoWPAN
  - Keine Routen außerhalb des LoWPAN
- Simple LoWPAN
  - Single Edge Router
- Extended LoWPAN
  - Multiple Edge Router mit gemeinsamen Backbone

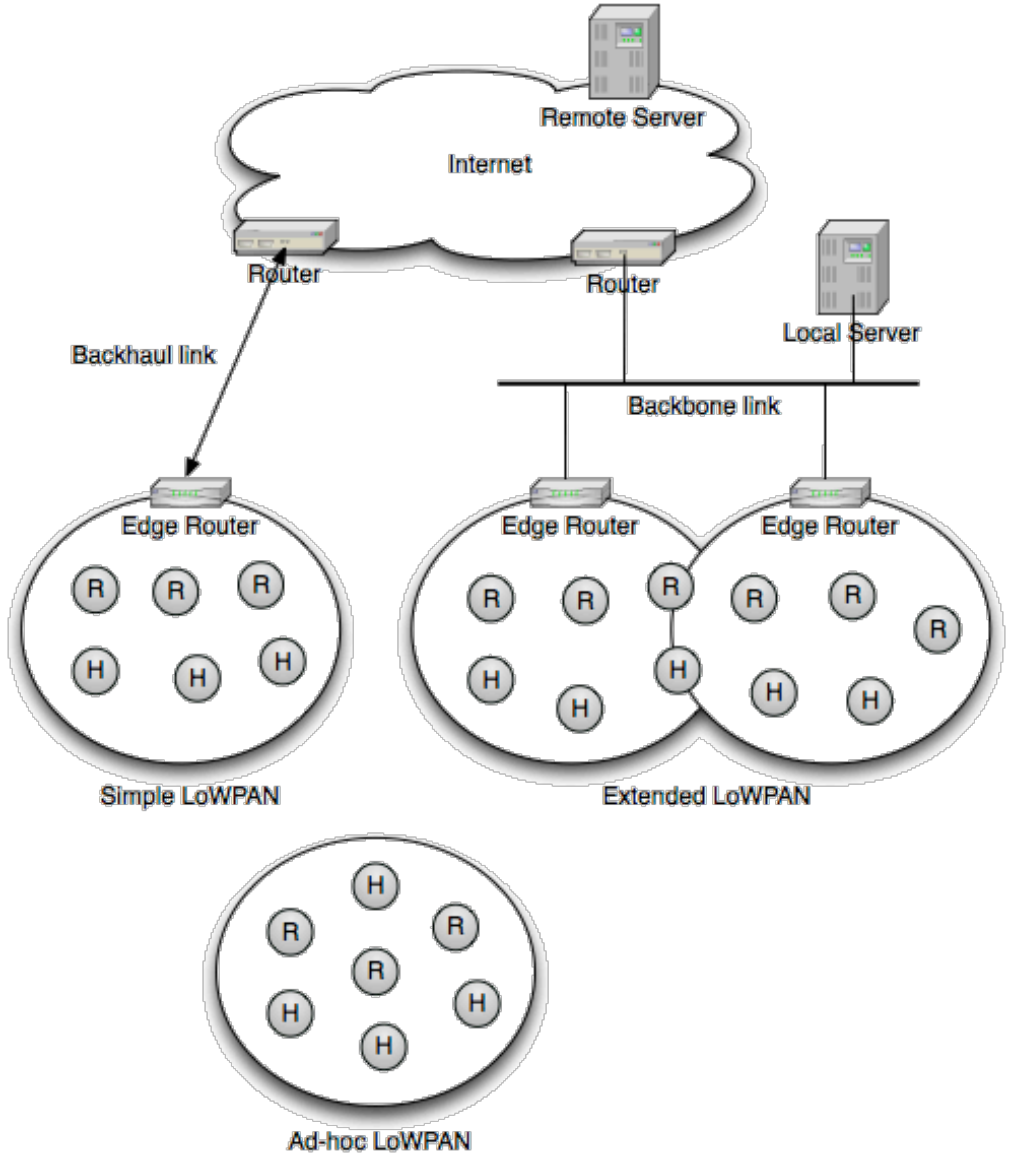


Quelle: http://6lowpan.net

LoWPAN Netztopologien

LoWPANs sind Stub-Netzwerke:

- Ad-hoc LoWPAN
  - Keine Routen außerhalb des LoWPAN
- Simple LoWPAN
  - Single Edge Router
- Extended LoWPAN
  - Multiple Edge Router mit gemeinsamen Backbone



Quelle: http://6lowpan.net

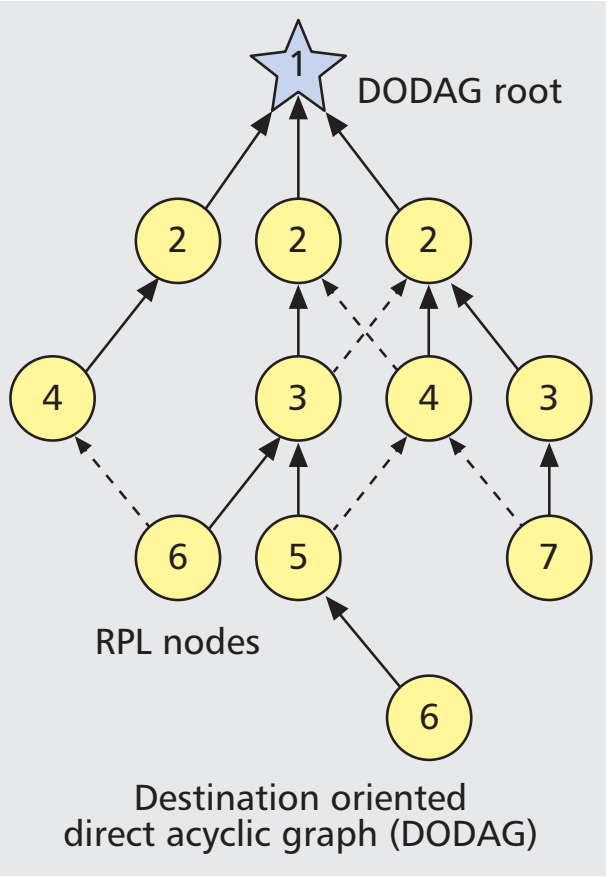
Routing in Low-Power Lossy Networks (RPL)
(RFC 6550)

Herausforderungen:

- Begrenzte Ressourcen (RAM, CPU, Batterie)
- Unstabile, verlustbehaftete Links mit geringer Datenrate

RPL Routing:

- Dynamisches Routingprotokoll
- Aufbau eines zielorientierten, gerichteten, azyklischen Graphs (DODAG)
- Minimale Information auf den Knoten
- Forwarding in den meisten Fällen über DODAG Root (Border Router)



Quelle: IEEE Communications Magazine, 04/2011

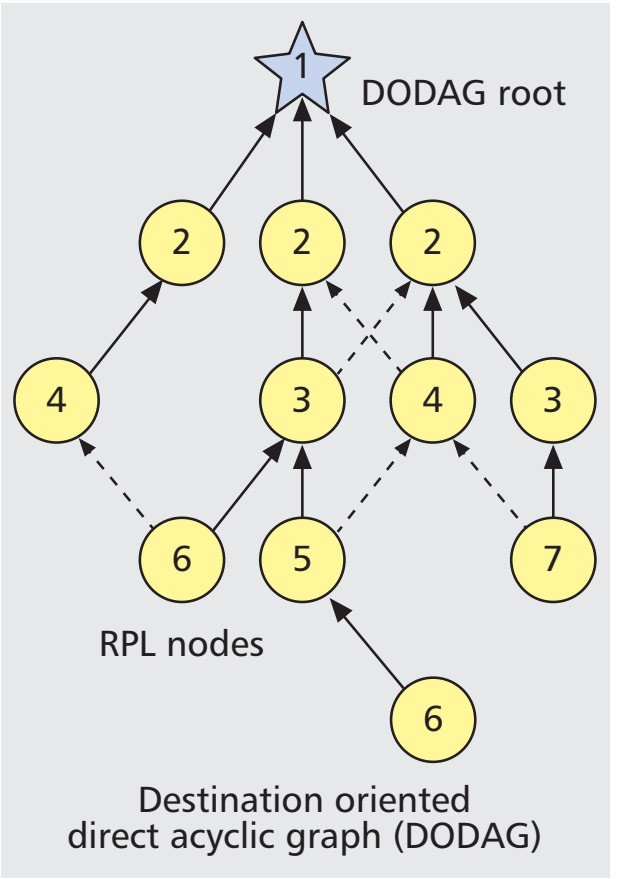
Routing in Low-Power Lossy Networks (RPL)
(RFC 6550)

Herausforderungen:

- Begrenzte Ressourcen (RAM, CPU, Batterie)
- Unstabile, verlustbehaftete Links mit geringer Datenrate

RPL Routing:

- Dynamisches Routingprotokoll
- Aufbau eines zielorientierten, gerichteten, azyklischen Graphs (DODAG)
- Minimale Information auf den Knoten
- Forwarding in den meisten Fällen über DODAG Root (Border Router)



Quelle: IEEE Communications Magazine, 04/2011

Quellen

- https://github.com/contiki-os/contiki/wiki
- http://contiki.sourceforge.net/docs/2.6/
- http://senstools.gforge.inria.fr/doku.php?id=contiki

$\Rightarrow$ alle Beispiele (inklusive Sender Quellcodes für Linux) sind unter https://github.com/szehl/contiki_tutorial zu finden.