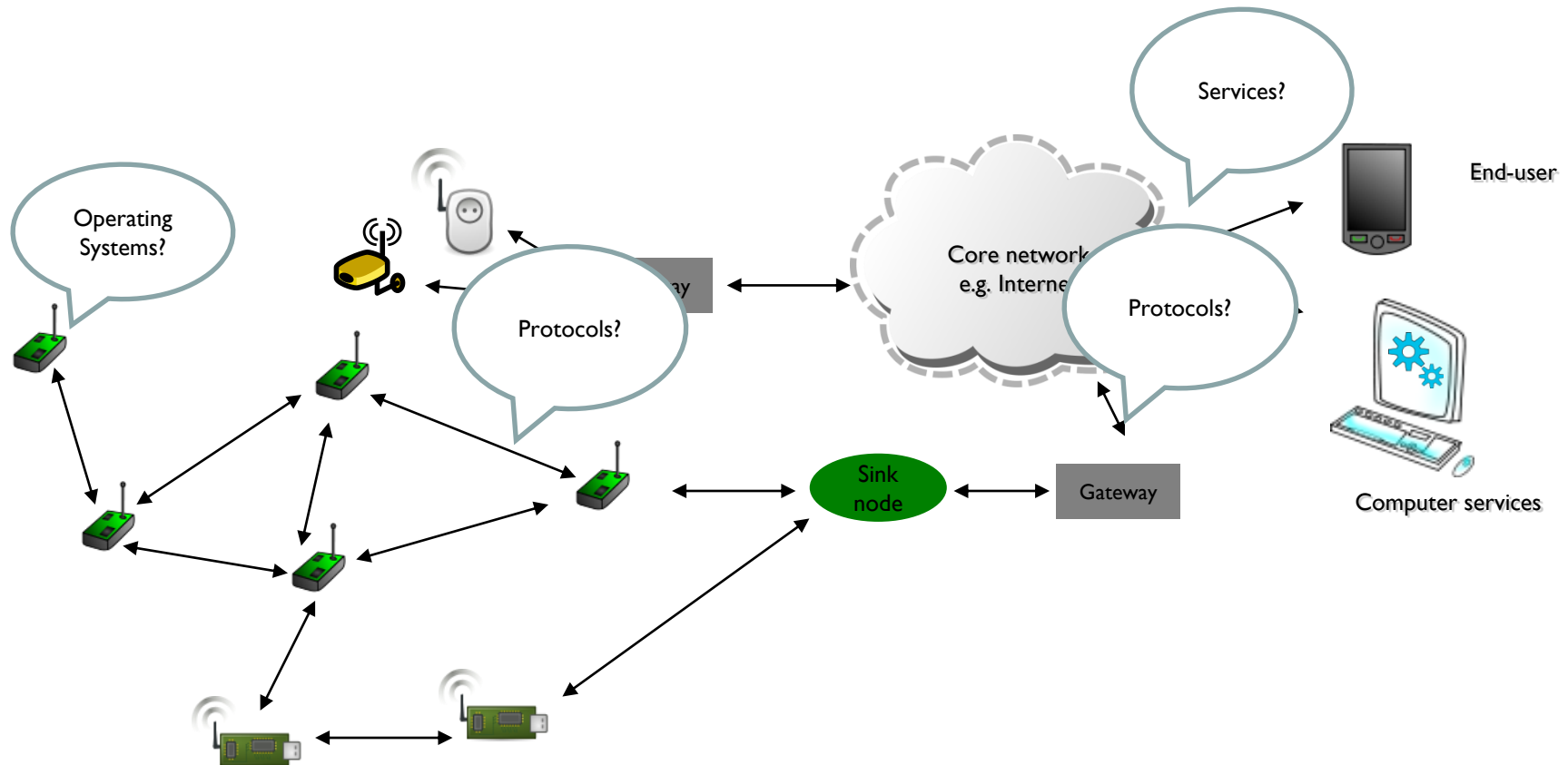


# Wireless Sensor (and Actuator) Networks



- The networks typically run on constrained devices
- Consist of one or more sensors (and actuators), could be different type of sensors (or actuators)

# Nodes and Applications in WSNs

- Sensor Networks consist of nodes with different capabilities.
  - Large number of heterogeneous sensor nodes
  - Spread over a physical location
  - It includes physical sensing, data processing and networking
- In ad-hoc networks, sensors can **join** and **leave** due to mobility, failure etc.
- Data can be processed in-network or it can be directly communicated to the endpoints.

# Types of nodes

- Sensor nodes
  - Low power
  - Consist of sensing device, memory, processor and radio
  - Resource-constrained
- Sink nodes
  - Another sensor node or a different wireless node
  - Normally more powerful/better resources
- Gateway
  - A more powerful node
  - Connection to core network
  - Could consist service representation, cache/storage, discovery and other functions

# Types of applications

- Event detection
  - Reporting occurrences of events
  - Reporting abnormalities and changes
  - Could require collaboration of other nearby or remote nodes
  - Event definition and classification is an issue
- Periodic measurements
  - Sensors periodically measure and report the observation and measurement data
  - Reporting period is application dependent
- Approximation and pattern detection
  - Sending messages along the boundaries of patterns in both space/time
- Tracking
  - When the source of an event is mobile
  - Sending event updates with location information

# Requirements and challenges

## – Fault tolerance

- The nodes can get damaged, run out of power, the wireless communication between two nodes can be interrupted, etc.
- To tolerate node failures, redundant deployments can be necessary.

## – Lifetime

- The nodes could have a limited energy supply;
- Sometimes replacing the energy sources is not practical (e.g. underwater deployment, large/remote field deployments).
- Energy efficient operation can be a necessity.

# Requirements and challenges – Continued

- Scalability
  - A WSN can consists of a large number of nodes
  - The employed architectures and protocols should scale to these numbers.
- Wide range of densities
  - Density of the network can vary
  - Different applications can have different node densities
  - Density does not need to be homogeneous in the entire network and network should adapt to such variations.

# Requirements and challenges – Continued

## – Programmability

- Nodes should be flexible and their tasks could change
- The programmes should be also changeable during operation.

## – Maintainability

- WSN and environment of a WSN can change;
- The system should be adaptable to the changes.
- The operational parameters can change to choose different trade-offs (e.g. to provide lower quality when energy efficiency is more important)



# Operating Systems

- An Operating System (OS) in an embedded system is a thin software that resides between the node's hardware and the application layer.
- OS provides basic programming abstractions to the application developer.
- The main task of the OS is to enable applications to interact with hardware resources, to schedule and prioritise tasks and mediate between applications and services that try to use the memory resources.

# Features of the OS in embedded systems

- Memory management
- Power management
- File management
- Networking
- Providing programming environment and tools (commands, interpreters, compiler, etc.)
- Providing entry points to access sensitive resources such as writing to input components.
- Providing and supporting functional aspects such as scheduling, multi-threading, handling interrupts, memory allocations.

# Operating systems and run-time in constrained environments

- Need for energy efficiency
- The code is more restricted (compared to conventional operating systems) so a full-blown OS is not required.
  - An appropriate programming model
  - A clear way to structure a protocol stack
  - And support for energy management

# Threads and events

- A “**thread**” in a programming environment is the smallest sequence of programmed instructions that can be managed and run independently by a scheduler.
- An **event driven** program typically runs an event loop. It keeps waiting for for an event, e.g. input from internal alarms. When an event occurs, the program collects data about the event and dispatches the event to the event handler software to deal with it.

# Thread-based vs Event-based Programming

- In WSN it is important to support concurrent tasks; in particular tasks related to I/O systems.
- **Thread-based programming** uses multiple threads and a single address space.
- This way if a thread is blocked by an I/O operation, the thread can be suspended and other tasks can be executed in different threads.
- The programmer must protect shared data structures with locks, coordinate the execution of threads.
- The program written for multiple threading can be complex, can include bugs and may lead to deadlocks.

# Thread-based vs Event-based Programming- II

- The **event-based programming** uses **events** and **event handlers**.
- Event handlers are registered at the OS scheduler and are notified when a named event occurs.
- The OS kernel usually implements a loop function that polls for events and calls relevant event handlers when an event is occurred.
- An event is processed by an event handler until completion unless it reaches a blocking operation.
- In the case of reaching a blocking operation it registers a new call back and returns control to scheduler.

# Dynamic Programming

- There could be a requirement for programming some parts of an application (for example in a WSN environment) after deployment. For example:
  - The complete deployment setting and requirements may not be known prior to the deployment and as a result the network may not function optimally;
  - Application requirements and physical environment properties can change over time;
  - There could be bug fixes or update requirements while the network is still operating.
- In the above cases, manual replacement of software may not be feasible because of the large number of nodes in the network.

- If there is no clear separation between OS and the application dynamic programming can not be supported.
- In dynamic programming, OS should be able to receive the software updates and store it in the active memory.
- OS should make sure that this is an updated version.
- OS should remove the piece of software that needs to be updated from memory and should install and configure the new version.



# Sensor Network Programming

- Sensor Network programming can be: **node centric** or it can be **application centric**.
- **Node-centric** approaches focus on development of a software for nodes (on a per-node level).
- **Application-centric** approaches focus on developing software for a part or all of the network as one entity.
- The **application centric** programming will require collaboration among different nodes in the network for collection, dissemination, analysis and/or processing of the generated and collected data.
- While in **node centric** programming the main focus is on developing a software on a per-node level.

# IoT Operating System #2: FreeRTOS

[KERNEL](#)[LIBRARIES](#)[RESOURCES](#)[COMMUNITY](#)[PARTNERS](#)[SUPPORT](#)[Download FreeRTOS](#)

Formally verifying FreeRTOS IPCs

Watch Nathan Chong describe his work to formally verify FreeRTOS queues

Watch on YouTube

## FreeRTOS™

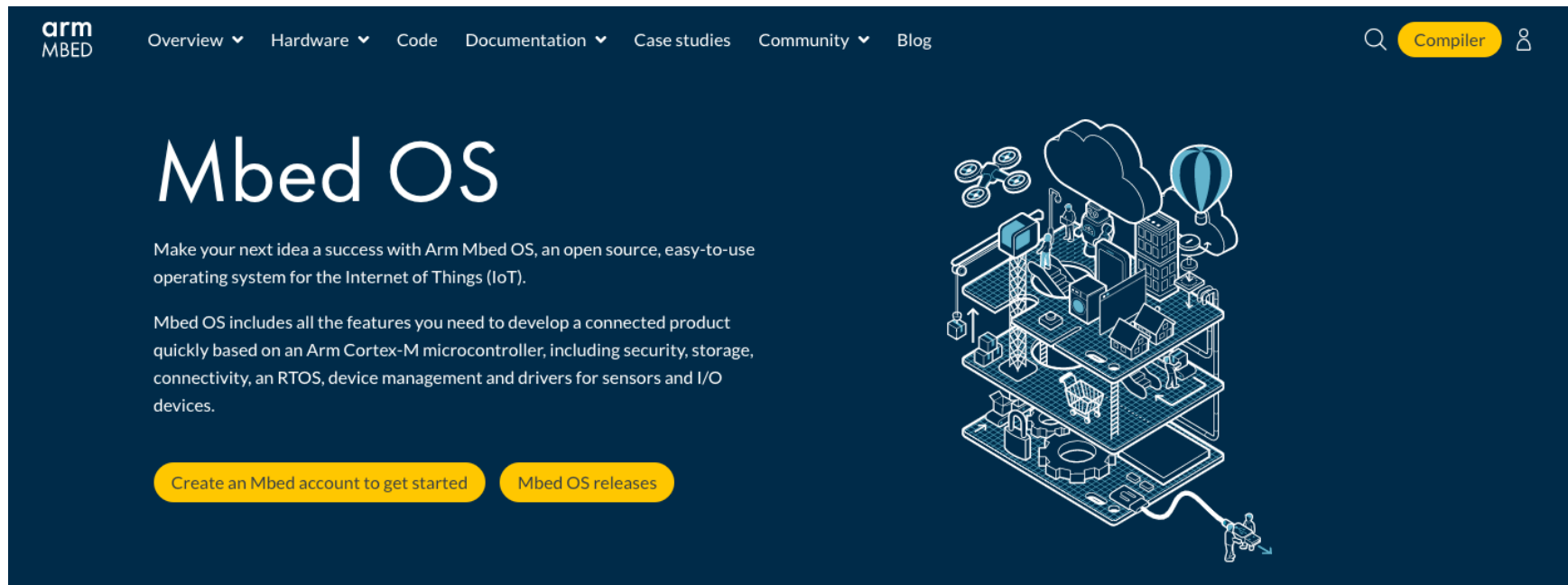
### Real-time operating system for microcontrollers

Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

[Download FreeRTOS](#)[Getting Started](#)

<https://ubidots.com/blog/iot-operating-systems/>

# IoT Operating System #3: Mbed OS

This is a screenshot of the Mbed OS website. The page has a dark blue background. At the top left is the "arm MBED" logo. A navigation bar contains links for Overview, Hardware, Code, Documentation, Case studies, Community, and Blog, each with a dropdown arrow. On the right, there is a search icon, a yellow "Compiler" button, and a user profile icon. The main heading "Mbed OS" is in large white font. Below it, a paragraph describes the OS as an open source, easy-to-use system for IoT. Another paragraph lists features like security, storage, connectivity, RTOS, and device management. Two yellow buttons are at the bottom: "Create an Mbed account to get started" and "Mbed OS releases". On the right side, there is a white line-art illustration of a multi-layered IoT system. The layers include a drone, a cloud, a hot air balloon, a city skyline, a shopping cart, and a person, all interconnected by lines representing data flow.

# IoT Operating System #4: [MicroPython](#)



MicroPython

[FORUM](#)

[DOCS](#)

[QUICK-REF](#)

[DOWNLOAD](#)

[STORE](#)

[CONTACT](#)

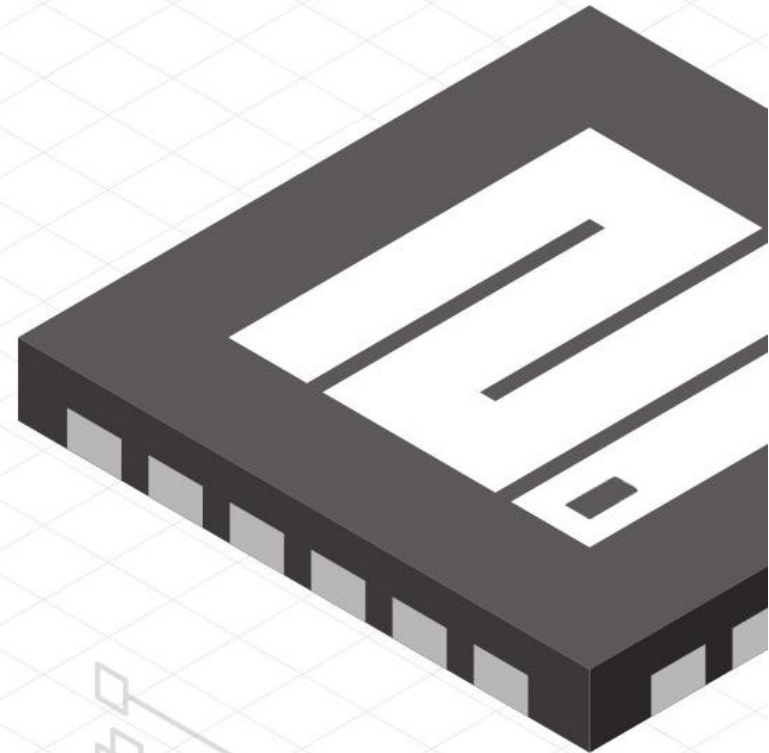
## MicroPython

MicroPython is a lean and efficient implementation of the **Python 3** programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments.

The MicroPython **pyboard** is a compact electronic circuit board that runs MicroPython on the bare metal, giving you a low-level Python operating system that can be used to control all kinds of electronic projects.

MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM.

MicroPython aims to be as compatible with normal Python as possible to allow you to transfer code with ease from the desktop to a microcontroller or embedded system.



[TEST DRIVE A PYBOARD](#)

[BUY A PYBOARD](#)

[USE MICROPYTHON ONLINE](#)

# IoT Operating System #5: Embedded Linux



CANONICAL

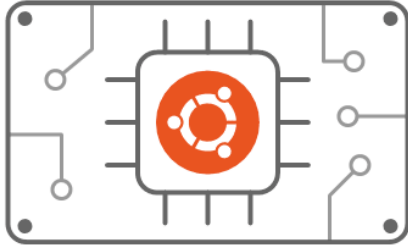
ubuntu® Enterprise ▾ Developer ▾ Community ▾ Download ▾

We are hiring Products ▾

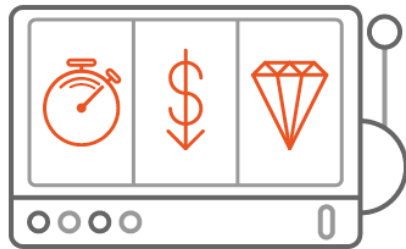
Search 🔍 Sign in

## Embedded Linux 2.0

It's time for a better developer experience.  
More reliable updates. Much better security.



Developers Apps Hardware



Faster, cheaper, better. Pick any 3.

- ✓ Reduce the time to design, develop and launch your devices.
- ✓ Lower the cost of development and maintenance.
- ✓ Raise the quality and security of every component.

# IoT Operating System #6: [RIOT](#)



# IoT Operating System #7: TinyOS

- “TinyOS is an open source operating system designed for low-power wireless devices, such as those used in sensor networks .”
- TinyOS applications are developed using nesC
- nesC is a dialect of the C language that is optimised for the memory limits of sensor networks.



# IoT Operating System: #8: Windows 10 IoT



Windows IoT

Explore ▾

Products ▾

Docs

Downloads

Community ▾

Support

Dashboard

All Microsoft ▾

Search 🔍

Sign in 

## Windows 10 IoT


Windows 10 IoT is a member of the Windows 10 product family that brings enterprise-class power, security, and manageability to the Internet of Things. Today, there are over 10,000 Windows IoT partners from the Edge to the Azure Cloud.

GET STARTED >





# IoT Operating System: #9: OpenWrt

OpenWrt  
WIRELESS FREEDOM

Search

Q

🔧

🇬🇧

Log In

You are here / [🏠](#) / [Downloads](#)

### Learn about OpenWrt

- [Supported devices](#)
- [Packages](#)
- [Downloads](#)**
- [Documentation](#)
  - [Quick start guide](#)
  - [User guide](#)
  - [Developer guide](#)
- [Security](#)
- [FAQ](#)
- [Forum](#)

### Contributing



- [Submitting patches](#)
- [Reporting bugs](#)
- [Contributing to wiki](#)

## Downloads

### Browse the OpenWrt/LEDE firmware repository





These links take you to the Downloads directory for the current hardware, grouped by processor type of the devices.

OpenWrt/LEDE software has two distinct branches: a stable **Release** build that is suitable for production use, and a **Development** build that contains an ever-evolving set of enhancements.

 <a href="#">Stable Release builds</a>	 <a href="#">Development Snapshot builds</a>
The <b>Release</b> builds have had significant testing. Use them for production, or for your home where your family will rely on a functioning router. <a href="#">More...</a>	Get the latest with a <b>Development</b> build. These contain the latest technology, but may not work well, or at all. Be prepared to supply bug reports, etc. <a href="#">More...</a>

#### Table of Contents

- Downloads
  - [Browse the OpenWrt/LEDE firmware repository](#)
  - [Download OpenWrt/LEDE firmware specific for your device](#)
  - [Get additional software packages](#)
  - [Build your own firmware](#)
  - [Assemble your own firmware](#)
  - [Build your own packages](#)
  - [Buildbot activity](#)
  - [Source code: Git repositories](#)
  - [Source code: GitHub mirrors](#)
  - Mirrors
    - [How to mirror](#)

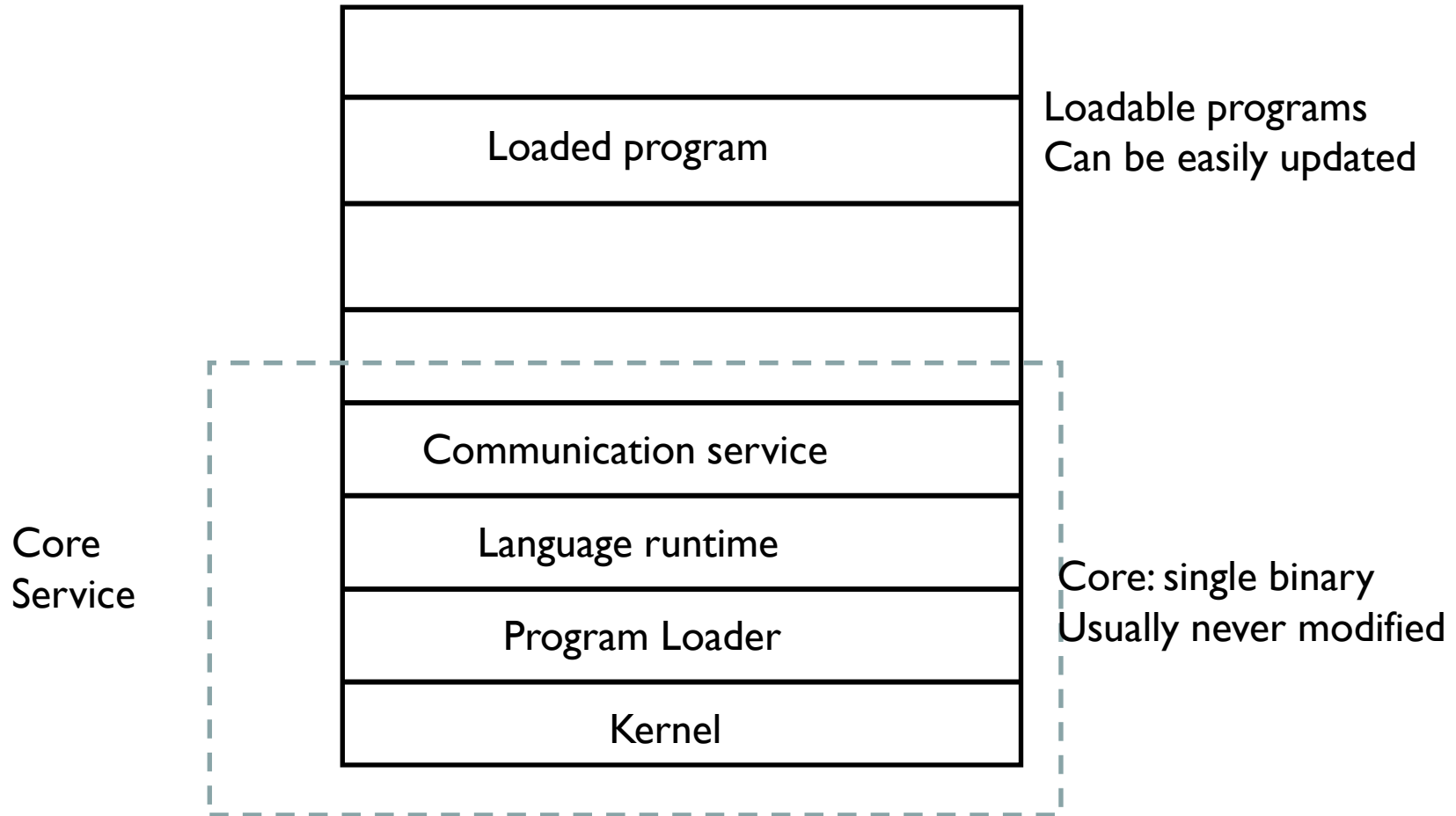


- Contiki is an open source operating system for networked embedded systems and wireless sensor nodes.
- It is designed for microcontrollers with small amounts of memory.
- A typical Contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM.
- Contiki provides IP communications, both for IPv4 and IPv6.
  - It has an IPv6 stack that, combined with power-efficient radio mechanisms such as ContikiMAC, allows battery-operated devices to participate in IPv6 networking.
  - Contiki supports **6lowPAN** header compression and the **CoAP** application layer protocol.
    - We will study 6LowPAN and CoAP protocols later in this module.

# Contiki- Functional aspects

- It's kernel functions as an event-driven kernel; multithreading is supported by an application library. In this sense it is a hybrid OS.
- Contiki realises the separation of concern of the basic system support from the rest of the dynamically loadable and programmable services (called processes).
- The services communicate with each other through the kernel by posting events.
- The ContikiOS kernel does not provide any hardware abstraction; but it allows device drivers and application directly communicate with the hardware.
- Each Contiki service manages its own state in a private memory space and the kernel keeps a pointer to the process state.

# The Contiki OS



# Protothreads

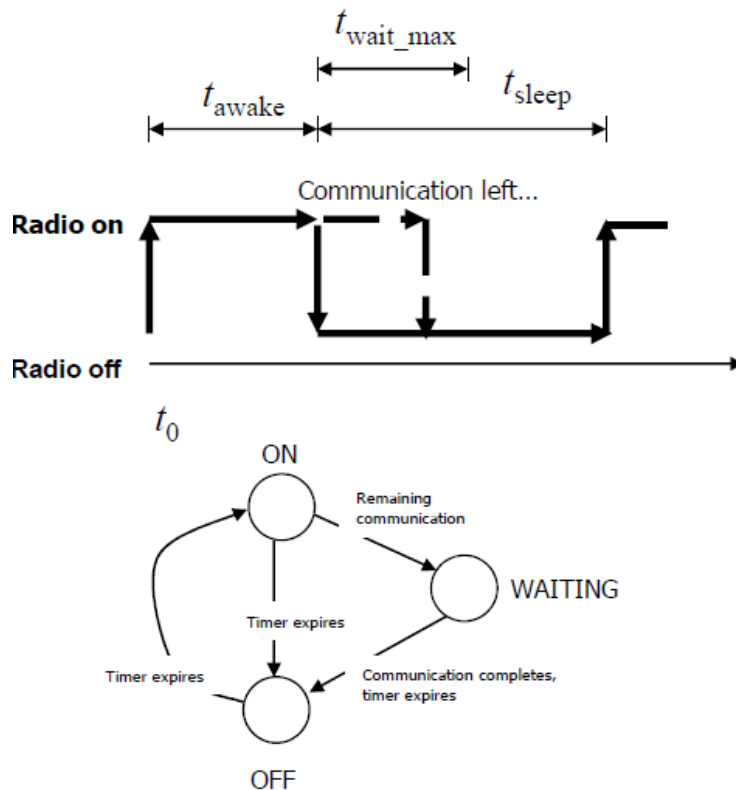
- Protothreads can be seen as lightweight (stackless) threads.
- They also can be seen as interruptible tasks in event-based programming.
- A protothread provides a conditional blocking “wait” statement which takes a conditional statement and blocks the protothread until the statement is evaluated true.
- By the time the protothread reaches the wait time if the conditional statement is true, it continues executing without any interruption.
- A protothread is invoked whenever a process receives a message from another process or a timer event.

# Protothreads- example

- For example consider a MAC protocol that turns off the radio subsystem on a periodic basis; but you want to make sure that the radio subsystem completes the communication before it goes to sleep state.
  1. *At  $t=t_0$  set the radio ON*
  2. *The radio remains on for a period of  $t_{awake}$  seconds*
  3. *Once  $t_{awake}$  is over, the radio has to be switched off, but any on-going communication needs to be completed.*
  4. *If there is an on-going communication, the MAC protocol will wait for a period,  $t_{wait\_max}$  before switching off the radio.*
  5. *If the communication is completed or the maximum wait time is over, then the radio will go off and will remain in the off state for a period of  $t_{sleep}$ .*
  6. *The process is repeated.*

# Radio sleep cycle code with events

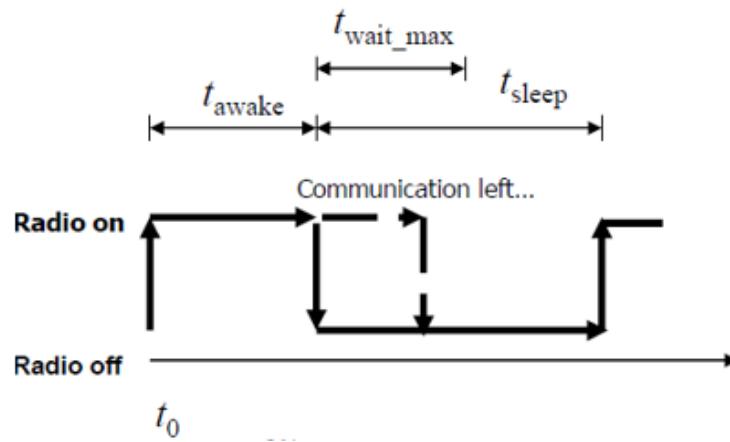
Event driven code can be messy and complex



```
enum {ON, WAITING, OFF} state;

void eventhandler() {
    if(state == ON) {
        if(expired(timer)) {
            timer = t_sleep;
            if(!comm_complete()) {
                state = WAITING;
                wait_timer = t_wait_max;
            } else {
                radio_off();
                state = OFF;
            }
        }
    } else if(state == WAITING) {
        if(comm_complete() ||
            expired(wait_timer)) {
            state = OFF;
            radio_off();
        }
    } else if(state == OFF) {
        if(expired(timer)) {
            radio_on();
            state = ON;
            timer = t_awake;
        }
    }
}
```

# Radio sleep cycle with Protothreads



```
int protothread(struct pt *pt) {
    PT_BEGIN(pt);
    while(1) {
        radio_on();
        timer = t_await;
        PT_WAIT_UNTIL(pt, expired(timer));
        timer = t_sleep;
        if(!comm_complete()) {
            wait_timer = t_wait_max;
            PT_WAIT_UNTIL(pt, comm_complete()
                          || expired(wait_timer));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, expired(timer));
    }
    PT_END(pt);
}
```



## Contiki process

- Contiki processes implement their own version of protothreads, that allow processes to wait for incoming events.
- The protothread statements used in Contiki processes are therefore slightly different than the pure protothread statements as presented in the previous section.

# Contiki Process

- Code in Contiki runs in either of two execution contexts: **cooperative** or **preemptive**.
- **Cooperative** code runs sequentially with respect to other cooperative code.
- **Preemptive** code temporarily stops the **cooperative** code.
- Contiki processes run in the **cooperative** context, whereas interrupts and real-time timers run in the **preemptive** context.
- All Contiki programs are processes. A process is a piece of code that is executed regularly by the Contiki system.
- Processes in Contiki are typically started when the system boots, or when a module that contains a process is loaded into the system. Processes run when something happens, such as a timer firing or an external event occurring.

# The Contiki code

- `#include "contiki.h"`
- `PROCESS(sample_process, "My sample process");`
- `AUTOSTART_PROCESSES(&sample_process);`
- `PROCESS_THREAD(sample_process, ev, data) {`
  - `PROCESS_BEGIN();`
    - `while(1) {`
      - `PROCESS_WAIT_EVENT();`
      - `}`
    - `PROCESS_END();`
  - `}`

Header files

Defines the name of the process

Defines the process will be started every time module is loaded

contains the process code

Event parameter; process can respond to events

Threads must have an end statement

process can receive data during an event

# Timers in Contiki OS

- The Contiki system provides a set of timer libraries that are used both by application programs and by the Contiki system itself.
- The timer libraries contain functionality for checking if a time period has passed, waking up the system from low power mode at scheduled times, and real-time task scheduling.
- The timers are also used by applications to let the system work with other things or enter low power mode for a time period before resuming execution.

# The Contiki Timer Modules

- The **timer** and **stimer** libraries provides the simplest form of timers and are used to check if a time period has passed. The applications need to ask the timers if they have expired.
- The **etimer** library provides event timers and are used to schedule events to Contiki processes after a period of time. They are used in Contiki processes to wait for a time period while the rest of the system can work or enter low power mode.

# The etimer Library

- The Contiki *etimer* library provides a timer mechanism that generate timed events. An event timer will post the event *PROCESS\_EVENT\_TIMER* to the process that set the timer when the event timer expires. The *etimer* library use *clock\_time()* in the clock module to get the current system time.
- An event timer is declared as a *struct etimer* and all access to the event timer is made by a pointer to the declared event timer.

# Setting the Etimer

- An event timer is always initialized by a call to *etimer\_set()* which sets the timer to expire the specified delay from current time.
- An event timer can be stopped by a call to *etimer\_stop()* which means it will be immediately expired without posting a timer event. *etimer\_expired()* is used to determine if the event timer has expired.

```
#include "sys/etimer.h"

PROCESS_THREAD(example_process, ev, data)
{
    static struct etimer et;
    PROCESS_BEGIN();

    /* Delay 1 second */
    etimer_set(&et, CLOCK_SECOND);

    while(1) {
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        /* Reset the etimer to trig again in 1 second */
        etimer_reset(&et);
        /* ... */
    }
    PROCESS_END();
}
```

# The Clock Module

- The function `clock_time()` returns the current system time in clock ticks.
- The number of clock ticks per second is platform dependent and is specified with the constant `CLOCK_SECOND`.
- The system time is specified as the platform dependent type `clock_time_t` and in most platforms this is a limited unsigned value which wraps around when getting too large.
- The clock module also provides a function `clock_seconds()` for **getting the system time in seconds as an unsigned long** and this time value can become much larger before it wraps around (136 years on MSP430 based platforms).
- **The system time starts from zero when the Contiki system starts.**



# Contiki code

```

• #include "contiki.h"
• PROCESS (sample_process, "My sample process");
• PROCESS (LED_process, "LED blink process");
•
• AUTOSTART_PROCESSES(&sample_process, &LED_process);
•
• PROCESS_THREAD(sample_process, ev, data) {
•     static struct etimer t;
•     static int c = 0;
•     PROCESS_BEGIN();
•
•     while(1) {
•         PROCESS_WAIT_EVENT();
•
•         if(ev == PROCESS_EVENT_TIMER) {
•             printf("Timer event #%i\n", c);
•             c++;
•             etimer_reset(&t);
•         }
•     }
•     PROCESS_END();
• }
•
• PROCESS_THREAD(LED_process, ev, data) {
•     static struct etimer timer;
•     static uint8_t leds_state = 0;
•     PROCESS_BEGIN();
•     while (1)
•     {
•         etimer_set(&timer, CLOCK_CONF_SECOND/4);
•         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
•         leds_off(0xFF);
•         leds_on(leds_state);
•         leds_state += 1;
•     }
•     PROCESS_END();
• }

```

Process thread names

Process thread 1

Process thread 2

## Typedefs

typedef uint8_t	<b>u8_t</b> The 8-bit unsigned data type.
typedef uint16_t	<b>u16_t</b> The 16-bit unsigned data type.
typedef uint32_t	<b>u32_t</b> The 32-bit unsigned data type.
typedef int32_t	<b>s32_t</b> The 32-bit signed data type.
typedef unsigned short	<b>uip_stats_t</b> The statistics data type.

# Some of the blocking macros in Contiki

- `PROCESS_WAIT_EVENT()`: yield and wait for an event to be posted to this process;
- `PROCESS_WAIT_EVENT_UNTIL(cond)`: yield and wait for an event, and for the `cond` condition to be true;
- `PROCESS_WAIT_UNTIL(cond)`: wait until the `cond` condition is true. If it is true at the instant of call, does not yield;
- `PROCESS_WAIT_WHILE(cond)`: wait while the `cond` condition is true. If it is false at the instant of call, does not yield;
- `PROCESS_PAUSE()`: post an event to itself and yield. This allows the kernel to execute other processes if there are events pending, and ensures continuation of this process afterward;
- `PROCESS_EXIT()`: terminate the process.

# Running Contiki on a Hardware

- Write your code
- Compile Contiki and the application
  - `make TARGET=XM1000 sample_process`
  - Make file

```
CONTIKI = ../..  
all: simple_process  
include $(CONTIKI)/Makefile.include
```

- If you plan to compile your code on the chosen platform more than once;
  - `make TARGET=XM1000 savetarget`
- Upload your code
  - `make simple_process.upload`
- Login to the device
  - `make login`

# Power saving mode in Contiki

- The Contiki Kernel **does not contain any explicit power saving** abstractions.
- However, Contiki allows the application specific parts of the system implement power saving mechanisms.
- To help an application decide when to power down the system, the event scheduler exposes the size of event queue; if there were not events scheduled, this information can be used to power down the processor.

# Communications and Network Protocol Support

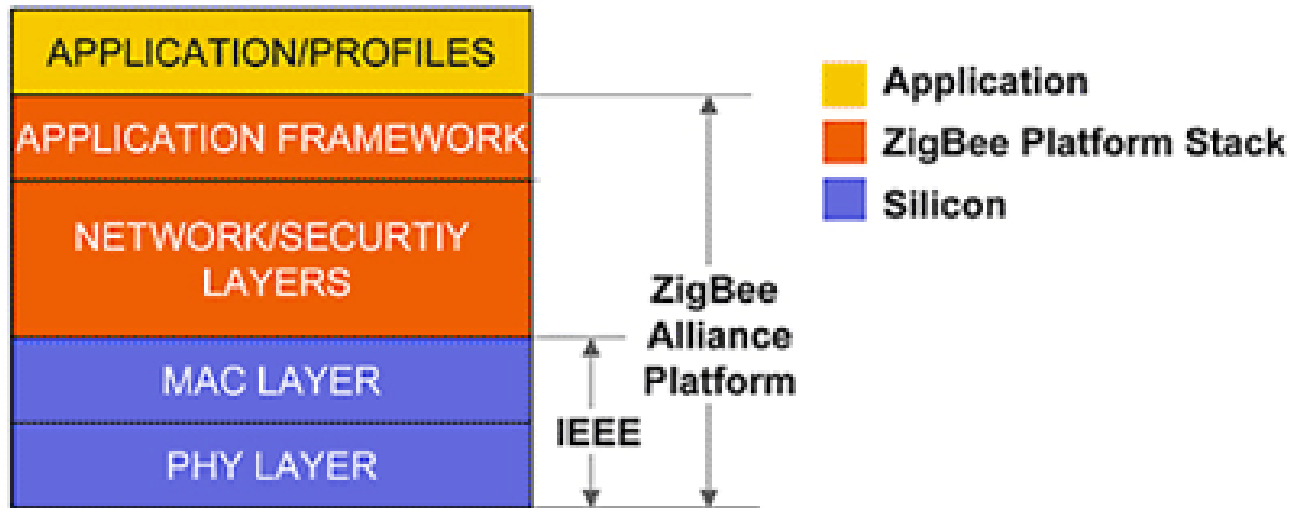
# Communication Protocols

- Wired
  - USB, Ethernet
- Wireless
  - Wifi, Bluetooth, ZigBee, IEEE 802.15.x
- Single-hop or multi-hop
  - Sink nodes, cluster heads...
- Point-to-Point or Point-to-Multi Point

- It is aimed to be a low cost, low power mesh network protocol.
- ZigBee operation range is in the industrial, scientific and medical radio bands;
- ZigBee's physical layer and media access control defined based on the IEEE 802.15.4 standard.
- ZigBee nodes can go from sleep to active mode in 30 ms or less, the latency can be low and in result the devices can be responsive, in particular compared to Bluetooth devices that wake-up time can be longer (typically around three seconds).

•

# ZigBee



	<u>BAND</u>	<u>COVERAGE</u>	<u>DATA RATE</u>	<u># OF CHANNEL(S)</u>
<b>2.4 GHz</b>	<b>ISM</b>	<b>Worldwide</b>	<b>250 kbps</b>	<b>16</b>
<b>868 MHz</b>		<b>Europe</b>	<b>20 kbps</b>	<b>1</b>
<b>915 MHz</b>	<b>ISM</b>	<b>Americas</b>	<b>40 kbps</b>	<b>10</b>



# Network protocols

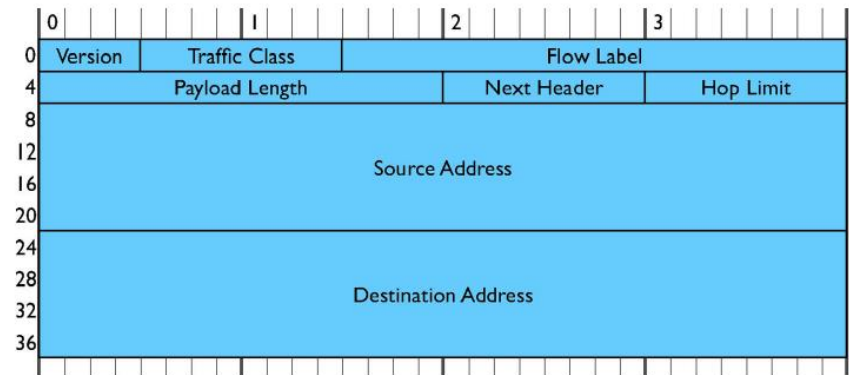
- The network (or OSI Layer 3 abstraction) provides an abstraction of the physical world.
- Communication protocols
  - Most of the IP-based communications are based on the IPV.4 (and often via gateway middleware solutions)
  - IP overhead makes it inefficient for embedded devices with low bit rate and constrained power.
  - However, IPv6.0 is increasingly being introduced for embedded devices
    - 6LowPAN

# IPv6 over Low power Wireless Personal Area Networks (6LoWPAN)

- 6LoWPAN typically includes devices that work together to connect the physical environment to real-world applications, e.g., wireless sensors.
- Small packet size
  - the maximum physical layer packet is 127 bytes
  - 81 octets ( $81 * 8$  bits) for data packets.
- Header compression
- Fragmentation and reassembly
  - 6LoWPAN defines a header encoding to support fragmentation when IPv6 datagrams do not fit within a single frame and compresses IPv6 headers to reduce header overhead.
- Support for both 16-bit short or IEEE 64-bit extended media access control addresses.
- Low bandwidth
  - Data rates of 250 kbps, 40 kbps, and 20 kbps for each of the currently defined physical layers (2.4 GHz, 915 MHz, and 868 MHz, respectively).

# 6LowPAN

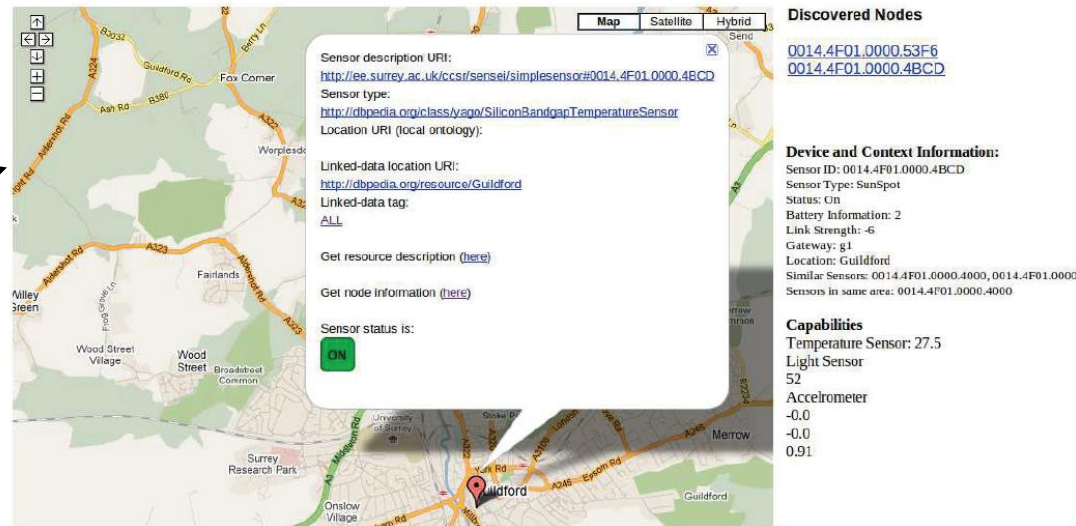
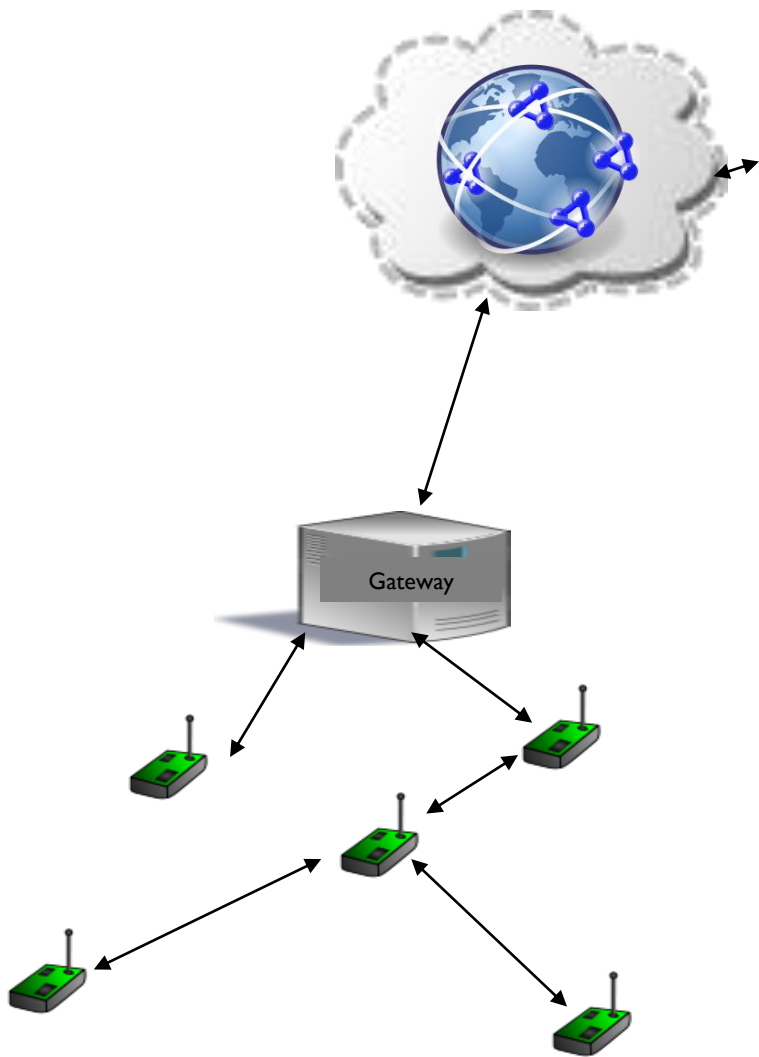
- IPv6 requires the link to carry a payload of up to 1280 Bytes.
- Low-power radio links often do not support such a large payload - **IEEE 802.15.4 frame only supports 127 Bytes of payload** and around 80 B in the worst case (with extended addressing and full security information).
- the IPv6 base header, as shown, is relatively large at **40 Bytes**.



## Using gateway and middleware

- It is unlikely that all IoT devices will be IP-enabled and/or will run an IP protocol stack.
- Gateway and middleware solutions can interfaces between low-level sensor island protocols and IP-based networks.
- The gateway can also provide other components such as QoS support, caching, mechanisms to address heterogeneity and interoperability issues.

# Gateway and IP networks



The screenshot shows a map application with a pop-up window displaying sensor information. The map shows a street view of Guildford, Surrey. The pop-up window contains the following information:

- Sensor description URI: <http://ee.surrey.ac.uk/ccsi/sensei/simplesensor#0014.4F01.0000.4BCD>
- Sensor type: <http://dbpedia.org/class/yago/SiliconBandgapTemperatureSensor>
- Location URI (local ontology): [ALL](#)
- Linked-data location URI: <http://dbpedia.org/resource/Guildford>
- Linked-data tag: [ALL](#)
- Get resource description ([here](#))
- Get node information ([here](#))
- Sensor status is: **ON**

On the right side of the screenshot, there is a section titled "Discovered Nodes" with two links: [0014.4F01.0000.53F6](#) and [0014.4F01.0000.4BCD](#). Below this is a section titled "Device and Context Information:" with the following details:

- Sensor ID: 0014.4F01.0000.4BCD
- Sensor Type: SunSpot
- Status: On
- Battery Information: 2
- Link Strength: -6
- Gateway: g1
- Location: Guildford
- Similar Sensors: 0014.4F01.0000.4000, 0014.4F01.0000
- Sensors in same area: 0014.4F01.0000.4000

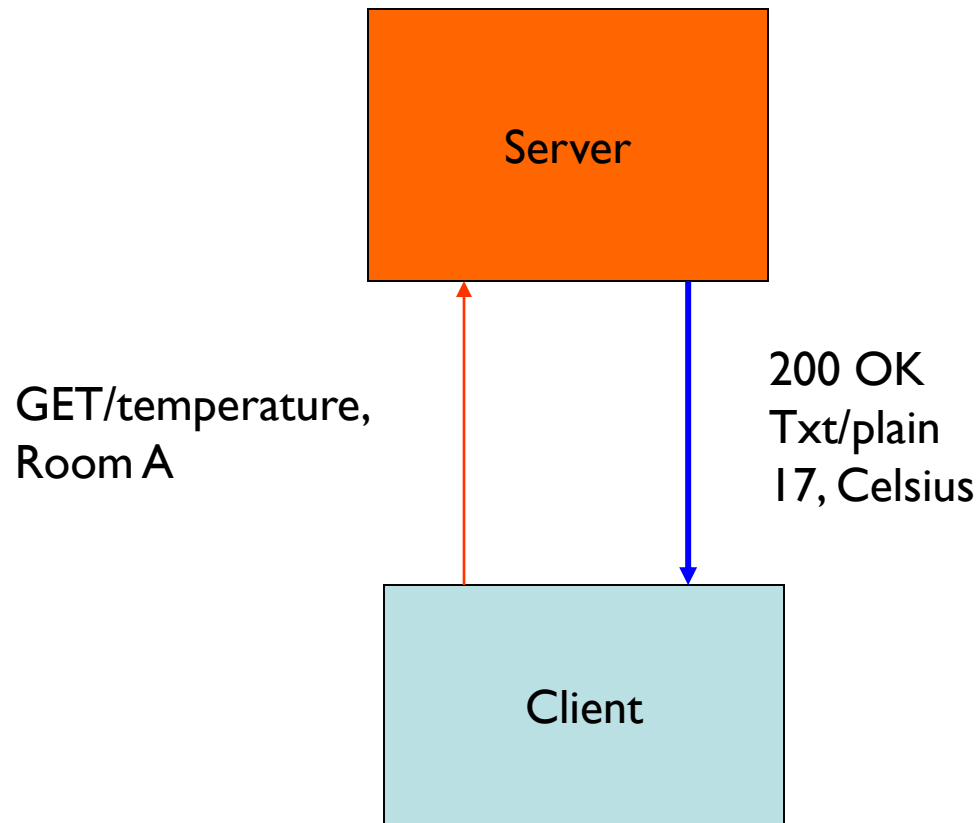
Below the device information is a section titled "Capabilities" with the following details:

- Temperature Sensor: 27.5
- Light Sensor: 52
- Accelerometer: -0.0, -0.0, 0.91

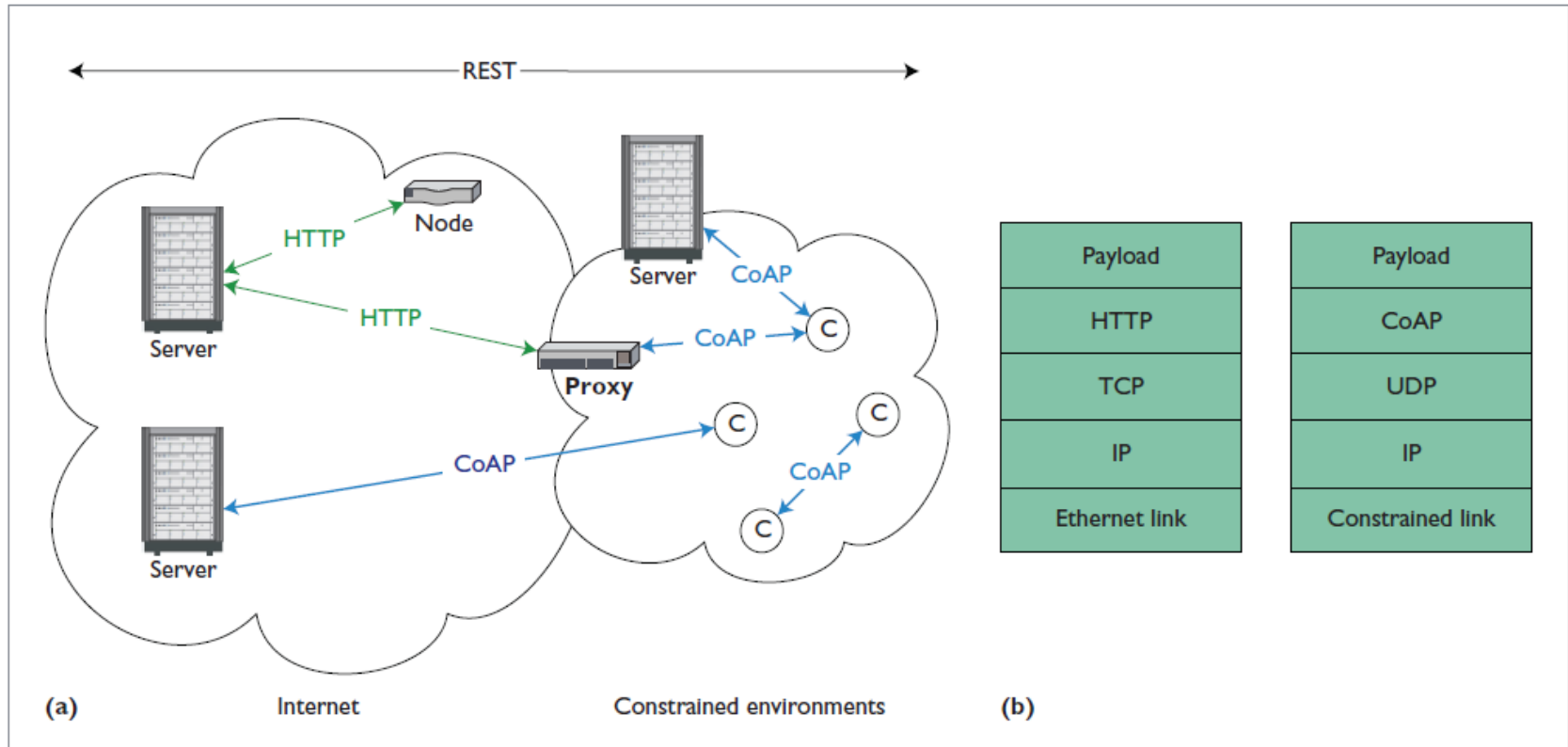
# Constrained Application Protocol (CoAP)

- CoAP is a transfer protocol for constrained nodes and networks.
- CoAP uses the Representational State Transfer (REST) architecture.
  - REST make information available as resources that are identified by URIs.
  - Applications communication by exchanging representation of these resources using a transfer protocol such as HTTP.
  - Clients access servicer controlled resources using synchronous request/response mechanisms.
    - Such as GET, PUT, POST and DELETE.
  - CoAp uses UDP instead of TCP and has a simple “message layer” for re-transmitting lost packets.
  - It also uses compression techniques.

# Constrained Application Protocol (CoAP)



# CoAP protocol stack and interactions



Implementing the Web architecture with HTTP and the Constrained Application Protocol (CoAP). (a) HTTP and CoAP work together across constrained and traditional Internet environments; (b) the CoAP protocol stack is similar to, but less complex than, the HTTP protocol stack.



## Further reading and examples

- Processes and protothreads: <https://github.com/contiki-os/contiki/wiki/Processes>
- Timers, <https://github.com/contiki-os/contiki/wiki/Timers>
- Examples: <https://github.com/contiki-os/contiki/tree/master/examples>
- Cooja simulator (if you are interested in advanced programming and simulation; this will not be a part of the exam):  
<https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja>

## Further reading (optional - not part of the exam)

- If you are interested in more advanced topics in Contiki OS programming:
  - The dynamic loader: <https://github.com/contiki-os/contiki/wiki/The-dynamic-loader>
  - Multithreading: <https://github.com/contiki-os/contiki/wiki/Multithreading>

## Optional- Linux Shell Commands

- If you would like to review the Linux shell commands this source can be helpful (the shell commands are useful during the lab sessions).
- Linux Shell Commands:  
[http://linuxcommand.org/lc3\\_lts0010.php](http://linuxcommand.org/lc3_lts0010.php)
- gedit Text File Editor:  
<https://help.gnome.org/users/gedit/stable/index.html.en>

# Acknowledgment

- Parts of the content is adapted from:
  - Waltenegus Dargie, Christian Poellabauer, “Fundamentals of Wireless Sensor Networks: Theory and Practice” (Wireless Communications and Mobile Computing), Wiley, 2010.
  - Holger Karl, Andreas Willig, “Protocols and Architectures for Wireless Sensor Networks”, Wiley, 2007.  
ISBN: 978-0-470-51923-3.

## **Acknowledgement:**

This course follows IoT module taught at University of Surrey.

# Questions?