Betriebssysteme für Sensorknoten im Vergleich

Florian Walter
Betreuer: Christoph Söllner
Seminar Sensorknoten: Betrieb, Netze und Anwendungen SS2010
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitektur
Fakultät für Informatik, Technische Universität München
Email: walterfl@in.tum.de

KURZFASSUNG

Die besonderen Anforderungen an Sensorknoten haben zur Entstehung von Betriebssystemen speziell für diesen Einsatzbereich geführt. Die Wahl des richtigen Systems für den eigenen Zweck bedarf einer fundierten Entscheidungsgrundlage. Vor diesem Hintergrund werden in der vorliegenden Abhandlung drei Betriebssysteme mit stark voneinander abweichenden Design-Konzepten vorgestellt: TinyOS, MANTIS OS und Contiki. Für jedes dieser Systeme wird ein kompakter Architekturüberblick einschließlich des resultierenden Laufzeitmodells präsentiert. Ebenso werden für den produktiven Einsatz relevante Aspekte wie Portierbarkeit auf neue Plattformen, Energieeffizienz, Toolchain im Entwicklungsprozess sowie Verfügbarkeit und Qualitiät von Dokumentation abgedeckt. Das klassische Konzept des Multithreadings und das Event-Konzept werden anhand von MANTIS OS und TinyOS verglichen.

Schlüsselworte

Sensorknoten, Betriebssystem, TinyOS, MANTIS OS, Contiki, Portabilität, Energieeffizienz, Dokumentation, Toolchain, event-driven, Multithreading, Protothread

1. EINLEITUNG

Die typischen Aufgaben eines Sensorknotens sind die Erfassung, Verarbeitung und Weiterleitung von Daten. Rechenleistung und Speicherkapazität der zugrundeliegenden Hardware sind beschränkt. Zudem werden die Geräte oft an schwer zugänglichen Orten fernab jeglicher Infrastruktur eingesetzt und über Batterien [25] und Solarzellen [36] mit Strom versorgt. Fortschritte in der Halbleiterfertigung werden sich daher eher in sinkender Energieaufnahme statt steigender Leistung der Sensorhardware niederschlagen [25, 28]. Eigens für Sensorknoten entwickelte Betriebssysteme sind speziell an diese knappen Hardware-Ressourcen angepasst. Sie nehmen dem Programmierer die Verwaltung von Hardware sowie Programmausführung ab und erleichtern dadurch die Anwendungsentwicklung.

In dieser Arbeit werden mit TinyOS, MANTIS OS und Contiki drei Betriebssysteme für Sensorknoten vorgestellt. Trotz ähnlicher Ziele verfolgen diese drei Systeme unterschiedliche Realisierungskonzepte. Wichtig für die erfolgreiche Entwicklung einer Anwendung ist die Wahl des für diese Anwendung am besten geeigneten Betriebssystems. Ebenso entscheidend ist die Unterstützung des Anwendungsentwicklers mit Dokumentation und ausgereiften Entwicklungswerkzeugen. Wissen über beide dieser Aspekte ist Voraussetzung für die begründete Entscheidung für oder gegen eines der genannten Systeme. Die vorliegende Abhandlung will dazu eine fundierte Basis liefern und stellt alle drei Betriebssysteme vor. Zu Beginn wird dargelegt, weshalb der Einsatz eines Betriebssystems auf Sensorknoten sinnvoll ist. In Abschnitt 3 wird TinyOS behandelt. Danach wird MANTIS OS genauer betrachtet. Während Tiny OS strikt den „event-driven“-Ansatz verfolgt, gilt MANTIS OS als Vertreter der Multithreading-Betriebssysteme. Beide Architekturparadigmen werden in Abschnitt 5 kurz verglichen. Das Betriebssystem Contiki wird in Abschnitt 6 als Kompromiss dieser Paradigmen vorgestellt. Die vorliegende Abhandlung schließt mit einer Zusammenfassung der Ergebnisse.

2. BETRIEBSSYSTEME AUF SENSORKNOTEN

2.1 Aufgaben eines Betriebssystems

Die Hauptaufgaben eines Betriebssystems sind die Verwaltung der Hardware und die Steuerung der Programmausführung [12]. Gerade eine effiziente Steuerung der Programmausführung ist auf Sensorknoten von großer Bedeutung. Denn um Energie zu sparen soll der Knoten einerseits möglichst früh in den Ruhezustand versetzt werden. Andererseits ist eine schnelle Reaktion auf eingehende Datenpakte notwendig. Nicht zeitkritische Berechnungen müssen dazu gegebenenfalls unterbrochen und später wieder aufgenommen werden.

2.2 Laufzeitsystem

Ein Betriebssystem setzt direkt auf der Hardware auf und stellt dem Benutzer eine komfortable Schnittstelle zur Verfügung, über die er die Hardware steuern kann. Diese Schnittstelle wird durch das Laufzeitsystem [12] realisiert. Anwendungen greifen nur indirekt über das Laufzeitsystem auf die Hardware zu. Ist also das Betriebssystem – und damit auch dessen Laufzeitsystem – für mehrere Hardware-Plattformen verfügbar, können bestehende Anwendungen beim Wechsel auf eine neue Plattform direkt übernommen werden.

3. TINY OS

TinyOS ist das älteste der hier betrachteten Betriebssysteme. Erste Versionen entstanden im Jahr 1999 an der University of California, Berkeley. Heute ist TinyOS ein Projekt der TinyOS Alliance und steht unter der BSD-Lizenz. Die aktuelle Version TinyOS 2 unterstützt eine Vielzahl von Prozessoren und Plattformen [44]. Im nächsten Abschnitt werden vier davon vorgestellt.

3.1 Unterstützte Hardware

TinyOS läuft auf den Mikrocontrollern der ATmega128-Serie [1] der Firma ATMEL und auf denen der MSP430-Serie [10] von Texas Instruments. Auch leistungsfähigere Prozessoren wie die aus der PXA27x-Serie von Marvell [7] werden unterstützt [40]. Zudem ist das Betriebssystem bereits an eine Vielzahl am Markt verfügbarer Sensorknoten angepasst [40]. Ein Beispiel ist der Sensorknoten MICAz [8] der Firma MEMSIC. Im Wesentlichen handelt es sich dabei um einen Mikrocontroller aus der ATmega128-Serie, an den ein 2,4 GHz Funkmodul des Typs Chipcon CC2420 angebunden ist [41].

3.2 Systemarchitektur

TinyOS ist kein Betriebssystem im klassischen Sinn. Vielmehr ist es eine Sammlung vieler einzelner Module, von denen jedes einen bestimmten Dienst anbietet. Im Kontext von TinyOS werden diese funktionalen Basiseinheiten *Komponenten*[1] genannt. Sie bilden die Basis für Architektur und Laufzeitmodell des Betriebssystems. Abbildung 1 gibt einen schematischen Überblick über den allgemeinen Aufbau einer Komponente.

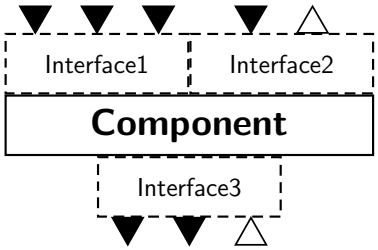

Abbildung 1: TinyOS-Komponente

3.2.1 Interfaces

Wie bereits erwähnt, kapselt jede Komponente bestimmte Dienste. Die Schnittstellen, über die diese zur Verfügung gestellt werden, heißen *Interfaces*. Im Beispiel aus Abbildung 1 bietet die Komponente `Component` Dienste über die Interfaces `Interface1` und `Interface2` an. Jedem Interface einer Komponente wird ein *Interface-Type* zugeordnet, der die Schnittstelle eindeutig beschreibt. Mit Interfaces werden nicht nur von Komponenten angebotene Dienste spezifiziert sondern auch Abhängigkeiten von anderen Diensten modelliert. Zur besseren Unterscheidung werden Interfaces letzteren Typs im Folgenden als *ausgehende Interfaces* bezeichnet. Im Beispiel aus Abbildung 1 benötigt die Komponente `Component` Zugriff auf eine andere Komponente, die `Interface3` implementiert.

3.2.2 Komponenten & Wiring

Wie das Betriebssystem selbst werden auch Anwendungen als Komponente oder als Sammlung von Komponenten entwickelt. Es gibt zwei Klassen von Komponenten: *Module*[2] bilden die kleinsten Einheiten im System und stellen eine Implementierung für die gekapselte Funktionalität bereit. *Konfigurationen*[3] verknüpfen mehrere Komponenten zu einer größeren Komponente. Ein lauffähiges Betriebssystem entsteht durch Verbindung aller notwendigen Komponenten über deren Interfaces. Der Bauplan, der angibt, wie die einzelnen Teile des Systems zu verknüpfen sind, heißt *wiring specification*. Dank dieses Bauplans werden immer nur unbedingt notwendige Komponenten in das Betriebssystem eingebunden.

3.3 Laufzeitmodell

Eng mit der Architektur verbunden ist das Laufzeitmodell von TinyOS. Wie bereits zu Beginn erwähnt, ist TinyOS kein typisches Multithreading-Betriebssystem, sondern arbeitet auf Basis von Events[4]. Als weitere Besonderheit wird die dynamische Allokation von Speicher während der Laufzeit nicht unterstützt [28].

3.3.1 Commands, Events und Tasks

Die Ausführung des Betriebssystems wird durch drei Grundprimitive gesteuert, die jeder Komponente zur Verfügung stehen und die im Folgenden genauer beschrieben werden.

- Über ausgehende Interfaces können *Commands* an andere Komponenten abgesetzt werden.
- *Events* werden von der Komponente gesendet, die einen Dienst zur Verfügung stellt, und werden über ausgehende Interfaces empfangen.
- Für Berechnungen, die ausschließlich innerhalb einer Komponente stattfinden, kennt TinyOS das Konzept der *Tasks*. Tasks repräsentieren Rechenaufträge für den Prozessor.

Letztlich handelt es sich bei Commands und Events um Funktionaufrufe [28]. Der Interface-Type enthält die Header der Command- und Event-Funktionen. Commands müssen von der Komponente implementiert werden, die einen Dienst anbietet und Events von der, die den Dienst nutzt.

3.3.2 Laufzeitverhalten

Die gerade beschriebenen Grundprimitive bestimmen maßgeblich das Laufzeitverhalten von TinyOS, welches nun dargestellt wird. Dienstanfragen, wie zum Beispiel die Erfassung eines Messwertes oder die Ausführung einer bestimmten Berechnung, werden als Command an die dafür bestimmte Komponente übermittelt. Anfragen, die wenig Rechenaufwand erfordern, können direkt in der Handler-Funktion des Commands bearbeitet werden. Umfangreichere Rechenaufgaben werden in Tasks ausgelagert. Jeder Task wird in eine zentrale Warteschlange eingereiht. Ein nicht-präemptiver Scheduler arbeitet diese Warteschlange gemäß einer First-Come-First-Served-Semantik ab. Eine Priorisierung von Tasks ist somit nicht möglich. Nur Commands, Events und Interrupt-Handler können einen aktiven Task unterbrechen. Mit Events werden von Tasks berechnete Ergebnisse übertragen oder Ereignisse wie Interrupts signalisiert.

[1] engl.: components
[2] engl.: modules
[3] engl.: configurations
[4] Dieses Architekturmodell heißt *event-driven*.

3.4 Beispiel

Die gerade beschriebenen Mechanismen werden nun anhand einer einfachen Anwendung [23] illustriert. Ziel ist die Entwicklung einer Diebstahlsicherung für Sensorknoten. Abbildung 2 zeigt alle beteiligten Komponenten und die Verknüpfung ihrer Interfaces, also die *wiring specification*. Commands eines Interfaces werden als schwarze und Events als weiße Pfeile dargestellt. `MainC` ist eine Konfiguration, die neben

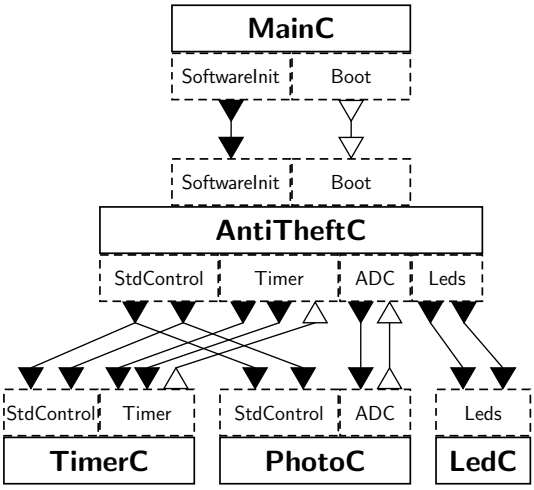

Abbildung 2: Diebstahlsicherung

dem Scheduler für den Systemstart notwendige Komponenten enthält [37]. Die Interfaces `SoftwareInit` und `Boot` steuern die Initialisierung der Komponente `AntiTheftC`. Diese wiederum steuert die angebundenen Sensoren über das Interface `StdControl`. Erfahrungsgemäß stecken Diebe gestohlene Sensorknoten in ihre Tasche. Der Messwert des durch `PhotoC` gesteuerten Helligkeitssensors fällt dann unter einen bestimmtem Schwellwert. Die Helligkeit wird daher regelmäßig gemessen. Die Fälligkeit der nächsten Messung wird durch `TimerC` signalisiert. `AntiTheftC` fragt daraufhin über das Interface `ADC` die aktuellen Sensordaten ab und vergleicht sie mit dem Schwellwert. Wurde ein Diebstahl erkannt, wird über das Interface `Leds` eine LED am Sensorknoten aktiviert, die den lichtscheuen Dieb abschrecken soll. Andernfalls wird keine Aktion ausgeführt oder die LED bei erfolgreicher Abschreckung gegebenenfalls deaktiviert.

3.5 Portabilität

Bei der großen Anzahl verfügbarer Sensorplattformen ist die Portabilität des Betriebssystems und der dafür geschriebenen Anwendungen von großer Bedeutung. Durch Kapselung von Diensten in Komponenten [28] können TinyOS selbst und für TinyOS entwickelte Anwendungen leicht an andere Plattformen angepasst werden. Denn bei der Portierung einer TinyOS-Anwendung müssen nur Komponenten mit direktem Hardwarezugriff neu implementiert werden. Solange die neue Implementierung die gleichen Interfaces wie die ursprüngliche Komponente bereitstellt, kann die Anwendung einfach durch Austausch der entsprechenden Komponenten an die neue Hardware-Plattform angepasst werden. Die durch das Interface-Konzept erwirkte strikte Trennung von Dienst und der Implementierung des Dienstes erlaubt zudem eine unkomplizierte Integration neuer Hardwarefunktionen in die bestehende Anwendung. Beispielsweise kann eine alte Komponente, die einen Kryptographiealgorithmus in Software realisiert, durch eine neue Komponente mit gleichen Interfaces ersetzt werden, die auf die Kryptographieeinheit der neuen Sensorhardware zurückgreift [28].

3.6 Energieeffizienz

TinyOS sieht mehrere Mechanismen für das Energie-Management vor. Mikrocontroller und Peripheriegeräte werden getrennt behandelt [39].

3.6.1 Mikrocontroller

Der Betriebszustand des Mikrocontrollers wird durch den Scheduler verwaltet. Sobald sich keine Tasks in der Warteschlange befinden, wird der Mikrocontroller in einen der verfügbaren Energiesparmodi versetzt. Die Berechnung und Aktivierung des am besten geeigneten Sparmodus erfolgt in der plattformspezifischen Komponente `McuSleepC` [38]. Optional können beliebige Komponenten die Wahl tieferer Sparmodi durch das Setzen einer unteren Schranke unterbinden.

3.6.2 Peripheriegeräte

Im Gegensatz zu Prozessoren kennen Peripheriegeräte meist lediglich die Zustände „An“ und „Aus“. Den Wechsel zwischen diesen Zuständen steuern die Anwendungen selbst[5] über Interfaces wie `StdControl` [39]. Beim Zugriff mehrerer Komponenten auf ein Gerät kann dieses Verfahren zu Konflikten führen. Für solche Szenarios können in TinyOS Richtlinien für die Steuerung des Geräts realisiert werden[6]. Die Komponente `PowerManager` setzt Richtlinien für das Energiemanagement durch. Wird ein Gerät von keiner anderen Komponente benötigt, wird es dem `PowerManger` zugeteilt. Dieser kann dann den Betriebszustand dieses Geräts anpassen. Sobald eine Komponente auf das Gerät zugreift, gibt der `PowerManager` die Kontrolle wieder ab.

3.7 Toolchain

3.7.1 nesC

TinyOS baut auf einer eigenen Programmiersprache namens *nesC* auf. nesC erweitert die Programmiersprache C um spezielle Konstrukte für das Architekturmodell von TinyOS. Allerdings werden weder die dynamische Allokation von Speicher noch Funktionspointer unterstützt [28]. Abbildung 3 zeigt die *wiring specification* der Beispielanwendung `AntiTheftApp` aus Abschnitt 3.4. Die Diebstahlsicherung verknüpft mehrere Komponenten und ist daher eine Konfiguration. Im Block **`configuration`** werden, falls vorhanden, die Interfaces der Konfiguration aufgeführt. Im Beispiel ist dieser Block leer. Im Abschnitt **`implementation`** werden zunächst alle Komponenten der Konfiguration aufgelistet. Anschließend werden die Interfaces dieser Komponenten mittels des Operators `->` verknüpft. Dabei steht auf der linken Seite des Operators die Komponente, die auf die Dienste der Komponente auf der rechten Seite des Operators zugreift. Der nesC-Compiler enthält Mechanismen zur Erkennung von Race Conditions[7] [24]. Zudem wird das Komponenten-Modell zur Programmoptimierung ausgenutzt [28].

[5] explizites Energiemanagement
[6] implizites Energiemanagent
[7] Konkurrierender Zugriff mehrerer Operationen auf die gemeinsam genutzte Variable führt zu unvorhersehbarem Ergebnis.

```
configuration AntiTheftApp {
}

implementation {
  components MainC, AntiTheftC, TimerC,
             PhotoC, LedC;

  MainC.SoftwareInit ->
                AntiTheftC.SoftwareInit;
  MainC.Boot -> AntiTheftC.Boot;
  AntiTheftC.StdControl -> TimerC.StdControl;
  AntiTheftC.Timer -> TimerC.Timer;
  AntiTheftC.StdControl -> PhotoC.StdControl;
  AntiTheftC.ADC -> PhotoC.ADC;
  AntiTheftC.Leds -> LedC.Leds;
}
```

Abbildung 3: Diebstahlsicherung in nesC

3.7.2 Buildsystem

Die Übersetzung des fertigen Codes und die Übertragung auf den Sensorknoten werden durch das GNU Buildsysem unterstützt. Ein mit TinyOS geliefertes Makefile steuert die Übersetzung des Quelltextes für die gewünschte Hardware-Plattform und die anschließende Übertragung der Software auf den Sensorknoten [42]. Noch bessere Unterstützung bei der Entwicklung von Anwendungen für TinyOS versprechen speziell auf dieses Betriebssystem abgestimmte Plugins für die IDE Eclipse. „NESCDT“ [9] erweitert Eclipse um Editierfunktionen für nesC. Das Plugin „Yeti 2“ [15] von der ETH Zürich unterstützt den gesamten Entwicklungsprozess einer TinyOS-Anwendung vom Schreiben des Codes bis hin zur Übertragung der fertigen Anwendung in den Flash-Speicher des Sensorknotens.

3.7.3 Simulator

Ein weiteres wichtiges Glied in der Toolchain von TinyOS ist der Simulator TOSSIM [43]. Mit TOSSIM können Sensornetzwerke mit hunderten von Knoten simuliert werden [27]. Durch Setzen einer Option des nesC-Compilers kann TinyOS für TOSSIM kompiliert werden. Dabei werden Komponenten mit Hardwarezugriff durch Komponenten von TOSSIM ersetzt. Je nach Anforderung können so auch ganze Mikrocontroller simuliert werden. Die eigentliche Simulation erfolgt gemäß einer übergebenen Konfiguration und umfasst auch eine drahtlose Netzwerkanbindung mit anpassbarer Fehlerrate [27].

3.8 Dokumentation

TinyOS und alle zugehörigen Tools sind umfassend dokumentiert. Zentraler Ausgangspunkt jeglicher Recherche ist die Projekt-Homepage [11]. Von dort aus gibt es Zugriff auf ein Dokumentations-Wiki. Eine Reihe von Tutorials ermöglicht den problemlosen Einstieg in TinyOS. Die Themen reichen von der Installation einer geeigneten Entwicklungsumgebung auf Windows, Mac OS X und Linux bis hin zur Nutzung des Simulators TOSSIM. TEPs[8] enthalten umfassende Beschreibungen der APIs von TinyOS. Das Git-Repository des Projekts enthält neben dem Quelltext des Betriebssystems zusätzlich viele Programmbeispiele. Insgesamt ist TinyOS hervorragend dokumentiert. Anwender mit grundlegenden Kenntnissen der Programmierung von Mikrocontrollern finden daher abgesehen vom neuen Programmiermodell einen leichten Einstieg.

[8]TinyOS Enhancement Proposals

4. MANTIS OS

Das nächste in dieser Abhandlung betrachtete Betriebssystem heißt MANTIS OS und ist im Jahr 2003 [14] an der University of Colorado at Boulder entwickelt worden. Die Software steht unter der BSD-Lizenz [32]. MANTIS steht als Akronym für „MultimodAl Networks of In-situ Sensors“.

4.1 Unterstützte Hardware

MANTIS OS unterstützt die Sensorplattformen MICA2, MICAz, und TelosB der Firma MEMSIC [32]. Der Sensorknoten MICA2 basiert ebenso wie der in Abschnitt 3.1 vorgestellte MICAz auf einem ATmega 128-Mikrocontroller. Bis auf das verwendete Funkmodul sind diese beiden Sensorknoten im Wesentlichen identisch. Der TelosB baut auf dem Mikrocontroller MSP430 von Texas Instruments auf. Neben einem Funkmodul und einer USB-Schnittstelle sind bereits mehrere Sensoren zum Messen von Lichteinstrahlung, Temperatur und Luftfeuchtigkeit auf dem Knoten integriert. Im Projektverzeichnis von MANTIS OS befindet sich zusätzlicher Code für die Unterstützung weiterer Hardware wie etwa der PXA27x-Prozessoren von Marvell [31].

4.2 Systemarchitektur

Das Design von MANTIS OS orientiert sich an klassischen UNIX-Betriebssystemen [13] und unterstützt präemptives Multitasking. Jede Anwendung ist daher ein eigener Thread. Freier Hauptspeicher wird als Heap verwaltet. Wie in TinyOS gibt es jedoch keine Möglichkeit zur dynamischen Allokation von Speicher durch den Programmierer. Lediglich das Betriebssystem selbst hat Zugriff auf den Heap [13]. Einen Überblick über die Architektur gibt Abbildung 4.

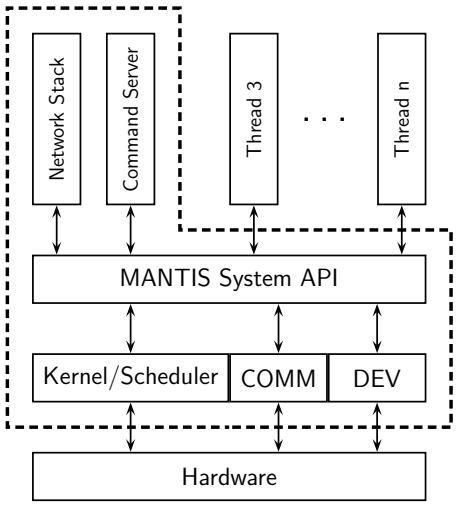

Abbildung 4: Systemarchitektur von MANTIS OS [13]

4.2.1 Kernel & Scheduler

MANTIS OS ist in mehreren Schichten organisiert. Anwendungsprogramme und der Netzwerk-Stack [13] laufen in Form von Threads auf der obersten Systemschicht. Die Kommunikation mit der darunterliegenden Schicht erfolgt über die API des Betriebssystems. Diese Schicht verfügt über direkten Hardwarezugriff und besteht aus Kernel und Treibern. Innerhalb des Kernels verwaltet ein Scheduler die Threads in einer statischen Tabelle [13]. Als Folge der festen Tabellengröße muss die maximale Anzahl gleichzeitiger laufender Threads bereits beim Kompilieren feststehen. Standardmäßig können maximal 10 Threads gleichzeitig gestartet sein [31]. In der Tabelle werden nur grundlegende Verwaltungsinformationen wie etwa die Priorität eines Threads gespeichert. Alle weiteren Kontextinformationen werden beim Threadwechsel auf dem Stack des verdrängten Threads gesichert. Der Stack eines Threads wird zur Laufzeit bei Bedarf auf dem Heap angelegt und nach Beendigung des Threads wieder freigegeben.

4.2.2 Synchronisation von Threads

Der unkontrollierte Zugriff mehrerer Threads auf gemeinsam genutzte Systemressourcen führt zu Race Conditions. MANTIS OS stellt zur Synchronisation solcher konkurrierender Threads Semaphore und Mutexe [13] zur Verfügung. Auf ein Semaphor wartende Threads werden in einer Liste innerhalb dieses Semaphors verwaltet [13]. Gleiches gilt für Mutexe [31].

4.2.3 Gerätetreiber

Gerätetreiber arbeiten wie der Kernel auf der untersten Schicht des Betriebssystems. Asynchrone Ein-/Ausgabe erfolgt über die COMM-Schicht, in der beispielsweise Funk- und USB-Verbindungen verwaltet werden. Diese Schicht verwaltet gemeinsam genutzte Datenpuffer, die in MANTIS OS `comBuf` genannt werden [13]. Anwendungen schreiben zu sendende Daten in einen solchen Puffer. Die COMM-Schicht nimmt diesen Puffer entgegen und übernimmt den eigentlichen Versand. Umgekehrt werden die empfangenen Daten ebenfalls in Puffern an die entsprechende Anwendung übergeben. Synchrone Ein-/Ausgabe wie zum Beispiel das Auslesen von Sensoren wird in der DEV-Schicht abgewickelt. Die API dieser Schicht lehnt sich an den POSIX-Standard an und bietet elementare Funktionen zur einheitlichen Ansteuerung der angebundenen Peripherie [13, 30]. Für jeden Gerätetreiber müssen lediglich vier Funktionen zur Gerätesteuerung sowie zum Lesen und Schreiben von Daten implementiert werden. MANTIS OS unterhält für jedes Gerät eine Tabelle, in der Zeiger auf diese vier Funktionen gespeichert sind. Interrupts werden grundsätzlich von den Gerätetreibern und nicht vom Kernel bearbeitet [13]. Lediglich der Thread-Wechsel durch den Scheduler wird durch einen Hardware-Timer ausgelöst. Software-Interrupts werden nicht unterstützt [13].

4.3 Laufzeitmodell

Jeder Thread in MANTIS OS ist einer von fünf Prioritätsstufen zugeteilt [13] und befindet sich immer in genau einem der folgenden Zustände [31]: Rechnend, rechenbereit, blockiert, schlafend. Der Scheduler verwaltet für jede Prioritätsstufe eine eigene Warteschlange. Threads gleicher Priorität werden in einem Round Robin-Verfahren abgearbeitet [34]. Eine Warteschlange wird nur dann bearbeitet, wenn die Warteschlangen aller höheren Prioritätsstufen leer sind. Die Priorität gibt also nicht an, in welchem Verhältnis die Rechenzeit unter den Threads aufgeteilt wird, sondern welche Threads zuerst vollständig abgearbeitet werden [33]. Gibt es keine rechenbereiten Threads, wird ein spezieller Idle-Thread ausgeführt [13]. Auf dessen Bedeutung wird bei der Diskussion der Energiesparmechanismen von MANTIS OS noch näher eingegangen. Ein Thread wechselt durch Aufruf der Funktion `mos_thread_sleep(uint16_t msecs)` für eine festgelegte Zeit in den Zustand „schlafend“. Dabei wird er aus seiner Warteschlange entfernt und in eine separate Warteschlange für schlafende Threads eingereiht [13]. Nach Ablauf der definierten Zeitspanne kehrt der Thread wieder in den Zustand „rechenbereit“ zurück und wechselt die Warteschlange. Der Zugriff auf ein belegtes Semaphor verläuft ähnlich: Der aufrufende Thread betritt die Warteschlange des Semaphors und geht in den Zustand „blockiert“ über. Bei Freigabe des Semaphors wird der erste Thread in dessen Warteschlange wieder in den Zustand „rechenbereit“ versetzt [31] und kehrt in seine ursprüngliche Warteschlange im Scheduler zurück.

4.4 Portabilität

Der Code von MANTIS OS ist gemäß der drei Teile der untersten Systemschicht unterteilt [31]. Architekturspezifischer Code ist von plattformunabhängigem Code getrennt [31]. Bei der Portierung des Kernels auf eine neue Prozessorarchitektur muss daher nur der architekturspezifische Teil des Codes neu implementiert werden. Jedoch enthält auch der eigentlich plattformunabhängige Teil des Kernels an einigen Stellen plattformspezifischen Code. Die Berücksichtigung dieser Sonderfälle und die Teils stark unterschiedliche Struktur des architekturspezifischen Codes dürfte die Portierung von MANTIS OS im Vergleich zum wesentlich klarer gegliederten TinyOS erschweren. Ein ähnliches Bild ergibt sich bei den bereits vorhandenen Treibern. Beispielsweise befinden sich die UART-Treiber für mehrere Sensorplattformen in einer einzelnen Datei. Erst der Präprozessor des Compilers entscheidet anhand von `ifdef`-Direktiven, welche Teile des Codes kompiliert werden sollen. Neue Treiber können dank der einfachen und standardisierten API jedoch leicht implementiert werden. Insgesamt wird deutlich, dass MANTIS OS nicht über die hohe Flexibilität über TinyOS verfügt. Trotzdem unterstützt die in vielen Fällen durchgesetzte Kapselung von hardwareabhängigem Code die Portierung des Systems auf neue Hardware-Plattformen.

4.5 Energieeffizienz

Der bereits erwähnte Idle-Thread ist maßgeblich für das Energiemanagement von MANTIS OS verantwortlich [13]. Da dieser Thread die niedrigste Priorität besitzt, wird er vom Scheduler nur dann ausgewählt, wenn die Warteschlangen aller anderen Prioritäten leer sind. Das ist dann der Fall, wenn kein Thread läuft oder alle Threads sich in den Zuständen „blockiert“ oder „schlafend“ befinden. Sobald der Idle-Thread läuft, werden die Parameter für den Energiesparmodus gewählt. Der Thread, der den Zustand „schlafend“ zuerst verlassen wird, bestimmt die Dauer. Zusätzlich wird abhängig von den Einstellungen der schlafenden Threads einer von zwei möglichen Energiesparmodi ausgewählt. Das differenzierte Energiemanagement von Mikrocontrollern wie dem ATmega128 [1], der sechs verschiedene Schlafzustän-

de anbietet, wird von MANTIS OS nicht unterstützt. Peripherie wird getrennt vom Prozessor über die Gerätetreiber durch die Anwendungen gesteuert.

4.6 Toolchain

4.6.1 Buildsystem

Wie bereits zu Beginn erwähnt, werden in der Architektur von MANTIS OS viele Konzepte aus UNIX-Systemen aufgegriffen. Das Betriebssystem und die API sind in der Programmiersprache C implementiert [13]. Das Thread-Konzept von MANTIS sowie die synchrone Ein-/Ausgabe über die DEV-Schicht orientieren sich am POSIX-Standard [13]. Dies erlaubt die Verwendung gängiger Entwicklungswerkzeuge. Zum Kompilieren des Codes unterstützt MANTIS OS SCons und das GNU Buildsystem [29].

4.6.2 Virtuelle Sensorknoten

Dank der Anlehnung der API von MANTIS OS an den POSIX-Standard ist zum Testen von Anwendungen kein spezieller Simulator notwendig. Stattdessen kann das gesamte Betriebssystem als UNIX- oder Windows-Anwendung kompiliert werden [13]. Aufrufe von Funktionen der MANTIS API werden werden über eine Zwischenschicht, den *„POSIX Shim Layer“*, in API-Aufrufe des zugrundeliegenden Betriebssystems übersetzt. Ein als UNIX-Anwendung laufendes MANTIS OS ist ein virtueller Sensorknoten, der in beliebige Sensornetzwerke eingebunden werden kann. So werden heterogene Netzwerke ermöglicht, die nicht allein auf reale Sensorknoten beschränkt sind. Abbildung 5 zeigt ein Beispiel für solch ein Netzwerk. In diesem Beispiel verbin-

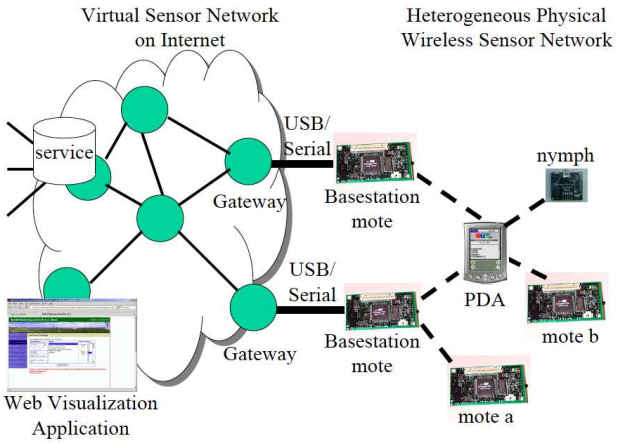

Abbildung 5: Heterogenes Sensornetzwerk mit virtuellen Sensorknoten im Internet [13]

den spezielle Bridge-Knoten das physische Sensornetzwerk über zwei Gateways mit den virtuellen Sensorknoten auf Servern im Internet. Virtuelle Sensorknoten sind nicht auf die API von MANTIS OS beschränkt. Dies ermöglicht die Einbindung von Anwendungen auf dem Host des virtuellen Knotens in das Netzwerk. Ein Nachteil dieser Flexibiltiät ist, dass aufgrund der Übersetzung der API im „Posix Shim Layer“ einige Teile von MANTIS OS wie etwa der Scheduler nicht getestet werden können [13]. Stattdessen muss in solchen Fällen auf Simulatoren wie Avrora [2] zurückgegriffen werden.

4.7 Dokumentation

Einen Großteil der Dokumentation zu MANTIS OS enthält ein ACM-Artikel aus dem Jahr 2005 [13]. Informationen über die Nutzung des Systems und über die Implementierung eigener Anwendungen liefert die Rubrik „Documentation“ auf der Homepage des Projekts [32]. Das SVN-Repository enthält eine genauere Beschreibung der API in Form des kommentierten Source-Codes. Zudem sind dort Beispielprogramme und C-Bibliotheken zu finden. Insgesamt ist die Dokumentation des Systems knapp. Die große Ähnlichkeit zu UNIX und die Verwendung der Programmiersprache C erlauben dennoch einen leichten Einstieg. Insbesondere muss anders als bei TinyOS kein neues Programmierkonzept erlernt werden.

5. EVENTS UND MULTITHREADING IM VERGLEICH

Wie gerade ausführlich erläutert wurde, unterscheidet sich das Event-Konzept von TinyOS deutlich vom Multithreading in MANTIS OS. Obwohl beide Konzepte gleich mächtig sind [26], ist die Entwicklung mit Threads oft leichter, da Programme bei der Verwendung von Events als Zustandsmaschinen programmiert werden müssen [21]. Zudem kann die rein sequentielle Abarbeitung von Tasks in TinyOS zu Problemen führen. Ein Beispiel ist der Empfang von Daten aus dem Netzwerk, die in einem Puffer zwischengespeichert werden [13]. Muss ein Task, der die Daten aus dem Puffer abarbeiten soll, längere Zeit auf einen anderen Task warten, kann der Puffer überlaufen. Ohne präemptives Multithreading lässt sich dieses Problem nur durch Elimination größerer Tasks umgehen. Dies ist jedoch nicht ohne genaues Verständnis der in einer Anwendung eingesetzten Algorithmen möglich und unter Umständen sehr aufwendig [13]. Die sequentielle Programmierung mit Threads ist meist einfacher. Ein genauerer Vergleich von TinyOS und MANTIS OS [16] hat ergeben, dass TinyOS weniger Speicher belegt und weniger Energie verbraucht als MANTIS OS. Andererseits reagiert MANTIS OS dank Multithreading bei umfangreicheren Rechenaufgaben schneller auf Anfragen aus dem Netzwerk als TinyOS.

6. CONTIKI

Contiki ist im Jahr 2003 am Swedish Institute of Computer Science entstanden. Wie die beiden anderen vorgestellten Betriebssysteme steht es unter der BSD-Lizenz. Aktuell ist die Version 2.4 verfügbar [5]. Einige der unterstützen Hardware-Plattformen werden im Folgenden vorgestellt.

6.1 Unterstützte Hardware

Contiki läuft auf den bereits bekannten ATmega 128-Mikrocontrollern und der MSP430-Serie von Texas Instruments. Unterstützt wird neben dem Sensorknoten MICAz von MEMSIC auch die an der Freien Universität Berlin entwickelte Plattform MSB430 [4]. Dieser Sensorknoten basiert auf einem MSP430-Prozessor und verfügt über ein Funkmodul sowie Sensoren für Beschleunigung, Temperatur und Luftfeuchtigkeit. Daten können auf einer SD-Karte gespeichert werden. Contiki wurde auch auf ältere Heimcomputer wie den Apple II oder den Commodore C64 portiert [17].

6.2 Systemarchitektur

Anders als die bisher vorgestellten Betriebssysteme verfügt Contiki über einen Programm-Lader, mit dem Anwendungen zur Laufzeit aus dem Speicher oder über ein Funkmodul geladen werden können [20]. Dies führt, wie Abbildung 6 zeigt, zur Gliederung des Betriebssystems in zwei Teile. Der

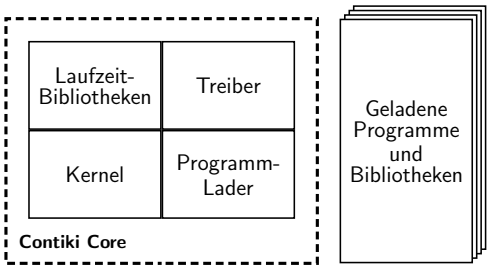

Abbildung 6: Systemarchitektur von Contiki [20]

Core ist ein Basissystem, das alle für den Betrieb wesentlichen Komponenten wie Kernel, Programm-Lader, Bibliotheken und Gerätetreiber enthält. Während des Betriebs können nur die geladenen Programme verändert werden. Änderungen am Core sind im Allgemeinen nicht vorgesehen und nur mit einem speziellen Bootloader möglich [20]. Die konkrete Aufteilung des Systems in Core und ladbare Programme wird beim Kompilieren des Systems entschieden und hängt von der Hardware-Plattform ab [20]. Gerätetreiber werden als Bibliotheken implementiert [20]. APIs für einige Geräteklassen wie etwa Funkmodule erlauben standardisierten Hardware-Zugriff [3]. Als einziges der hier vorgestellten Betriebssysteme unterstützt es die dynamische Allokation von Speicher [3].

6.3 Laufzeitmodell

Im Laufzeitmodell von Contiki werden die beiden Konzepte von MANTIS OS und TinyOS vereint. Einerseits besitzt das Betriebssystem einen Kernel, der Prozesse verwaltet. Andererseits basiert der Kernel auf dem Event-Konzept [20].

6.3.1 Events in Contiki

Contiki-Prozesse sind sogenannte *Protothreads* ohne eigenen Stack [4] und kommunizieren über *synchrone* und *asynchrone* Events mit anderen Prozessen. Eine Warteschlange im Kernel speichert alle eintreffenden asynchronen Events. Die Scheduling-Funktion des Kernels läuft nach Systemstart in einer Endlosschleife [3]. In jedem Durchlauf wird ein Event aus der Schlange entnommen und an den Empfänger-Prozess weitergeleitet. *Synchrone* Events werden ohne Umweg über die Warteschlange direkt an den Empfänger-Prozess zugestellt [20]. Die Abarbeitung eines Events in einem Prozess kann grundsätzlich nur durch Hardware-Interrupts unterbrochen werden [20]. Anders als in TinyOS werden größere Rechenaufträge nicht in Tasks ausgegliedert, sondern direkt im Prozess bearbeitet. Contiki stellt dazu mehrere Primitive zur Verfügung, mit denen die Berechnung gesteuert werden kann. Beispielsweise blockiert `PROCESS_WAIT_EVENT()` den aufrufenden Prozess, bis zum Eintreffen eines Events. Mit `PROCESS_WAIT_UNTIL(c)` wartet der Prozess, bis die Bedingung `c` erfüllt ist. Mit `PROCESS_YIELD()` wird ein Prozess unterbrochen. Im Gegensatz zum Komponentenmodell von TinyOS können Anwendungen mit Prototheads in sequentieller Form programmiert werden. Die meist komplizierte Aufteilung einer Berechnung in Commands, Tasks und Events entfällt [21].

6.3.2 Polling

Neben Events unterstützt Contiki *Polling*. Jeder Protothread kann zusätzlich zum Event-Handler einen Poll-Handler implementieren. Polls sind Events mit hoher Priorität. Sie sind besonders bei der Abarbeitung von Hardware-Interrupts wichtig, da Interrupts-Handler keine Events, sondern nur Polling-Anfragen absetzten dürfen [20]. Nach der Bearbeitung eines asynchronen Events testet der Scheduler ein Polling-Flag. Ist dieses Flag gesetzt, werden die Poll-Handler aller Prozesse aufgerufen, für die eine Polling-Anfrage vorliegt.

6.3.3 Multithreading

Über eine Bibliothek unterstützt Contiki neben Protothreads auch normale Threads mit eigenem Stack. Diese Threads können in einem Prozess erzeugt werden und werden nicht durch den Scheduler des Betriebssystems verwaltet. Stattdessen muss die Ausführung der Threads durch den Prozess selbst gesteuert werden [19].

6.4 Portabilität

Wie bei MANTIS OS ist in Contiki architekturspezifischer Code von plattformunabhängigem Code getrennt [3]. Jedoch ist diese Unterteilung bei Contiki deutlich konsequenter durchgesetzt. Die bei der Portierung anzupassenden Teile des Betriebssystems sind klar definiert: Bootprozess, architekturspezifische Teile des Programm-Laders und der Multithreading-Bibliothek [20]. Auch Treiber müssen auf die neue Plattform angepasst werden. Da Contiki nur für wenige Hardware-Komponenten standardisierte APIs enthält, muss der Entwickler eventuell auch Anwendungen an die Treiber der neuen Plattform anpassen.

6.5 Energieeffizienz

Contiki bietet keinerlei Unterstützung für das Energie-Management von Mikrocontroller und Peripherie. Stattdessen sollen die Anwendungen selbst entsprechende Mechanismen implementieren [20]. Die einzige Unterstützung, die Contiki dem Programmierer bei der Entwicklung von Energiesparmechanismen bietet, ist die Möglichkeit, die Anzahl der Events in der Warteschlange des Schedulers abzufragen [20].

6.6 Toolchain

6.6.1 Buildsystem

Contiki ist wie MANTIS OS in der Programmiersprache C geschrieben. Das Konzept der Protothreads erfordert wie bereits gezeigt spezielle Anweisungen. Anders als bei TinyOS werden die neuen Konstrukte nicht über eine Spracherweiterung von C sondern über Makros für den Präprozessor des Compilers realisiert. Abbildung 7 zeigt das Grundgerüst eines „Hello World“-Prozesses in Contiki. Als Makros realisierte Anweisungen enthalten nur Großbuchstaben. Die Verwendung von C erlaubt den Einsatz gängiger Editoren und Werkzeuge. Die Erstellung eines lauffähigen Systems wird durch das GNU Buildsystem unterstützt.

```
#include "contiki.h"
#include "stdio.h"

PROCESS(hello_world_process,
                "Hello world process");

PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
```

Abbildung 7: Aufbau eines Prozesses in Contiki [3]

6.6.2 Simulatoren

Zum Testen von Anwendungen enthält das Projekt-Verzeichnis von Contiki zwei Simulatoren. *MSPsim* [22] ist ein Simulator für Sensorknoten auf Basis des MSP430 von Texas Instruments, der Code für diesen Prozessor nativ ausführen kann. Neben dem Prozessor selbst werden auch Peripheriegeräte des Sensorknotens simuliert. Das Verhalten von Sensornetzen kann mit mit *COOJA* [35] getestet werden. COOJA kann entweder wie MSPsim die Hardware eines Knotens exakt nachbilden oder nur dessen Verhalten simulieren. Beides ist in einem simulierten Netzwerk zur gleichen Zeit möglich. Alle hier vorgestellten Tools sind auch in *Instant Contiki* [6] enthalten. Instant Contiki ist eine komplette Entwicklungsumgebung für Contiki die als Image für den VMWare Player heruntergeladen werden kann.

6.7 Dokumentation

Die gesamte technische Dokumentation des Betriebssystems ist über den Punkt „Documentation“ von der Projekthomepage [5] aus erreichbar. Anleitungen zu Installation und Benutzung sind ebenfalls direkt auf der Homepage verlinkt. Genaue technische Details zu Konzept und Realisierung von Contiki findet man in diesen Quellen jedoch nicht. Insbesondere gibt es keine Artikel, die einen Überblick über den Aufbau des Systems beschreiben. Der einzige vom Autor auffindbare Artikel, der Contiki näher beschreibt [20], ist veraltet. Beispielsweise wird dort noch das verworfene [18] Service-Konzept beschrieben. Insgesamt liegt der Schwerpunkt der Dokumentation auf der praktischen Anwendung von Contiki. Mit dem Quelltext werden viele Beispiele, Bibliotheken und fertige Anwendungen geliefert, die den Einstieg in die Programmierung erleichtern.

7. ZUSAMMENFASSUNG & FAZIT

In dieser Abhandlung wurden drei Betriebssysteme für Sensornetzwerke vorgestellt: TinyOS, Mantis OS und Contiki. Trotz vieler Gemeinsamkeiten setzen TinyOS, MANTIS OS und Conikti unterschiedliche Schwerpunkte.

7.1 TinyOS

TinyOS ist in erster Linie auf geringen Ressourcenverbrauch optimiert. Davon zeugt neben durchdachten Energiesparmechanismen vor allem das Event-Konzept, das die vom Betriebssystem beanspruchte Rechenzeit minimiert. Die aktive Weiterentwicklung des Systems durch die TinyOS Alliance macht TinyOS zusammen mit der Unterstützung einer Vielzahl von Sensorknoten zum attraktivsten System in diesem Vergleich. Die hervorragende Dokumentation macht das Betriebssystem auch für kommerzielle Anwendungen besonders interessant.

7.2 Contiki

Contiki versucht die Vorteile von Multithreading und Events in Protothreads zu vereinen. Damit fällt der Einstieg leichter als bei TinyOS, erfordert aber mehr Aufwand als bei MANTIS OS. Negativ fällt auf, dass keinerlei Maßnahmen zur Reduzierung des Energieverbrauchs implementiert sind. Unter den unterstützten Hardware-Plattformen finden sich nur wenige Sensorknoten. Die Dokumentation ist umfangreich, enthält aber keine Details zur Realisierung des Betriebssystems. Insgesamt nimmt Contiki so nur den zweiten Rang innerhalb dieses Vergleichs ein.

7.3 MANTIS OS

Die Architektur von MANTIS OS realisiert präemptives Multithreading. Dies führt zu einem im Vergleich zu TinyOS geringfügig höheren Ressourcenverbrauch. Vorteile ergeben sich vor allem bei rechenlastigen Anwendungen. Die große Ähnlichkeit zu typischen UNIX-Systemen erlaubt trotz sehr knapp gehaltener Dokumentation einen einfachen Einstieg in die Anwendungsentwicklung. Im Vergleich zum streng modular aufgebauten TinyOS ist MANTIS OS deutlich schlechter portierbar. Es gibt keine nennenswerte Entwicklergemeinschaft und die Weiterentwicklung scheint zu stagnieren. Vor allem deswegen belegt MANTIS OS in diesem Vergleich den letzten Platz.

8. LITERATUR

[1] ATMEL ATmega128A. http://www.atmel.com/dyn/products/product_card_v2.asp?PN=ATmega128A.

[2] Avrora. http://avrora.sourceforge.net/.

[3] Contiki 2.4 Source Code. http://www.sics.se/contiki/download.html.

[4] Contiki Dokumentation. http://www.sics.se/~adam/contiki/docs/.

[5] Contiki Projekt-Homepage. http://www.sics.se/contiki/.

[6] Instant Contiki. http://www.sics.se/contiki/instant-contiki.html.

[7] Marvell PXA-Prozessoren. http://www.marvell.com/products/processors/applications/pxa_family/.

[8] MEMSIC Wireless Modules. http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html.

[9] NESCDT. http://docs.tinyos.net/index.php/NESCDT.

[10] Texas Instruments MSP430. http://www.ti.com/ww/de/msp430.htm.

[11] TinyOS Homepage. http://www.tinyos.net/.

[12] U. Baumgarten and H. Siegert. *Betriebssysteme: Eine Einführung*. Oldenbourg, 6., überarb., aktualis. u. erw. A. edition, Dec. 2006.

[13] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis OS: An embedded multithreaded

operating system for wireless micro sensor platforms. In *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, page 2005, 2005.

[14] Q. Cao and T. Abdelzaher. The liteos operating system: Towards unix-like abstractions for wireless sensor networks.

[15] Distributed Computing Group, ETH Zürich. Yeti 2. `http://tos-ide.ethz.ch/wiki/index.php`.

[16] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan. A comprehensive experimental comparison of event driven and multi-threaded sensor node operating systems.

[17] A. Dunkels. About Contiki. `http://www.sics.se/contiki/about-contiki.html`.

[18] A. Dunkels. Contiki Changelog. `http://www.sics.se/contiki/changelog.html`.

[19] A. Dunkels. Using Multi-Threading in Contiki. `http://www.sics.se/contiki/developers/using-multi-threading-in-contiki.html`, 2007.

[20] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors, 2004.

[21] A. Dunkels and O. Schmidt. Protothreads - Lightweight, Stackless Threads in C, 2005.

[22] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Poster Abstract: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards.

[23] D. Gay. TinyOS 2.0: A wireless sensor network operating system. `http://hinrg.cs.jhu.edu/git/?p=tinyos-2.x.git;a=blob;f=apps/AntiTheft/tutorial-slides.pdf`, 2005.

[24] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.1 Language Reference Manual*. TinyOS.net, 2003.

[25] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47:41–46, 2004.

[26] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Operating Systems Review*, pages 3–19, 1979.

[27] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications, 2003.

[28] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[29] MANTIS Group. Building and running an example application. `http://mantisos.org/index/tiki-read_article.php%3FarticleId=6.html`.

[30] MANTIS Group. MANTIS Device Layer. `http://mantisos.org/index/tiki-index.php%3Fpage=ReferenceDevLayer.html`.

[31] MANTIS Group. MANTIS OS 1.0 Beta Source Code. `http://mantisos.org/index/tiki-read_article.php%3FarticleId=1&page=2.html`.

[32] MANTIS Group. MANTIS OS Projekt Homepage. `http://mantisos.org/`.

[33] MANTIS Group. MANTIS Thread Priorities. `http://mantisos.org/index/tiki-index.php%3Fpage=ReferenceThreadPriority.html`.

[34] MANTIS Group. MANTIS Threads. `http://mantisos.org/index/tiki-index.php%3Fpage=ReferenceThreadPriority.html`.

[35] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Crosslevelsensor network simulation with cooja. InLCN, 2006.

[36] W. K. Seah, Z. A. Eu, and H. P. Tan. Wireless sensor networks powered by ambient energy harvesting (WSN-HEAP)-Survey and challenges. In *1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009*, pages 1–5, 2009.

[37] TinyOS.net. TEP 107. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep107.html`.

[38] TinyOS.net. TEP 112. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep112.html`.

[39] TinyOS.net. TEP 115. `http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep115.html`.

[40] TinyOS.net. TinyOS Hardware-Unterstützung. `http://docs.tinyos.net/index.php/Platform_Hardware`.

[41] TinyOS.net. TinyOS MICAz-Unterstützung. `http://docs.tinyos.net/index.php/MicaZ`.

[42] TinyOS.net. TinyOS Toolchain. `http://docs.tinyos.net/index.php/TinyOS_Toolchain`.

[43] TinyOS.net. TOSSIM. `http://docs.tinyos.net/index.php/TOSSIM`.

[44] TinyOS.net. TinyOS git-Repository. `http://hinrg.cs.jhu.edu/git/?p=tinyos-2.x.git;a=tree;f=tos/chips`, Stand: 07.06.2010.