Systemsoftware in Sensornetzwerken

Bernhard Stähli
ETH Zürich – Departement Informatik

Abstract

Das Betriebssystem ist ein wesentlicher Bestandteil eines Sensornetzknotens. Ich stelle die verschiedenen Aspekte vor, die beim Design dieser Software in Betracht gezogen werden müssen. Insbesondere hat man in diesen Knoten mit sehr beschränkten Ressourcen zu kämpfen, was einige Auswirkungen auf die Architektur hat. Ich stelle TinyOS vor. Ein Betriebssystem, das versucht, den Anforderungen in einem Sensornetzwerk gerecht zu werden. Im Weiteren komme ich auf die Vor- und Nachteile von virtuellen Maschinen in einem Sensornetzwerk zu sprechen. Dies anhand der Implementation Maté.

1 Einführung

1.1 Aufgaben von Systemsoftware

Die Aufgabe von Systemsoftware ist es, die Ressourcen eines Rechners zu verwalten. Dazu gehören zum Beispiel CPU-Zeit, Hauptspeicher, Festplattenspeicher und Kommunikationsmittel. Es müssen konkurrierende Prozesse verwaltet werden und die Ressourcen unter ihnen fair und effizient aufgeteilt werden. Zudem muss die Systemsoftware eine stabile Basis für Anwendungsprogramme zur Verfügung stellen.

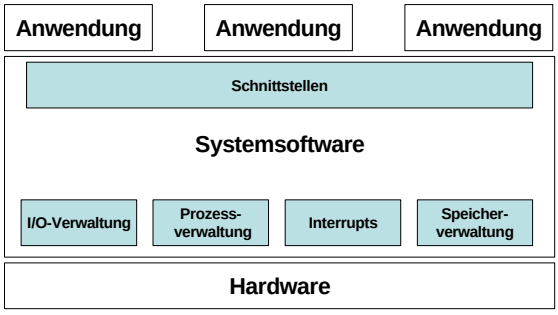

Abbildung 1: Schema eines Betriebssystems

1.2 Auf welcher Hardware muss die Systemsoftware laufen?

Im Wesentlichen ist die Hardware, die der Systemsoftware zur Verfügung steht, ziemlich spärlich ausgestattet. Am Beispiel eines Berkeley-Motes sieht das etwa folgendermassen aus: Es wird ein 4 MHz getakteter Prozessor eingesetzt. Er besitzt 32 Register, 8 kB Flashspeicher für den Programmcode und 512 Bytes Datenspeicher. Es gibt drei Energiezustände: Im „Idle"-Modus ist der Prozessor ausgeschaltet. Im „Powerdown"-Modus ist alles ausgeschaltet, ausser asynchrone Interruptus, die später das Aufwachen ermöglichen. Der

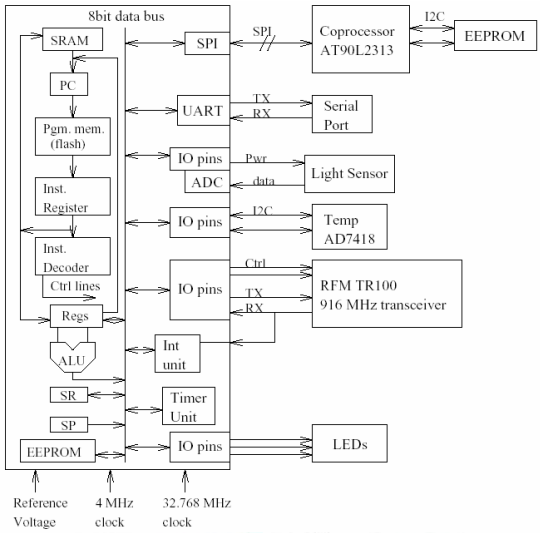

Abbildung 2: Hardware-Schema

„Powersave"-Modus entspricht dem „Powerdown"-Modus, belässt aber einen Timer am Laufen.

Über einen seriellen Bus können bis zu 8 externe Sensoren angeschlossen

werden. Die Sende- und Empfangseinheit ist die wichtigste Komponente. Es ist eine asynchrone Komponente, die bei 916.50 MHz betrieben wird. Die Bandbreite beträgt 19.2 kB/s. Die Kommunikationseinheit kann drei Zustände annehmen: Einen „Sende"-, einen „Empfangs"- und einen „Poweroff"-Zustand. Geplant ist auch ein Selbstüberwachungssystem, das es ermöglicht, Daten über den Zustand des Sensorknotens selbst zu erhalten. Zum Beispiel den Batterieladezustand oder die Empfangsstärke eines Radiosignals.

2 Spezielle Anforderungen an Systemsoftware in Sensornetzen

2.1 Geringer Speicherbedarf und Energieeffizienz

Die genannten Einschränkungen beeinflussen in wesentlichem Masse die Rechen-, Speicher- und Kommunikationsfähigkeit eines Sensorknotens. Obwohl Miniaturisierung und Energieeffizienz treibende Kräfte für die Hardwareindustrie sind, muss die Software gleichsam weiterentwickelt werden, um die stetig wachsenden Kapazitäten auch ausnutzen zu können. Dadurch, dass nur beschränkt Energie zur Verfügung steht, wird ein Prozessor eingesetzt, der etwas langsamer arbeitet, dafür aber weniger Energie verbraucht. Deshalb wird er auch so oft als möglich in einen energiesparenden Modus versetzt werden. Der Speicherplatz ist auch relativ beschränkt. In den Berkley-Motes sind nur etwa 3 kB für das Betriebssystem vorgesehen. Dies ist im Vergleich zu einem Desktopbetriebssystem nicht gerade viel.

2.2 Concurrency

Die Hauptaufgabe eines Sensorknotens besteht ja darin, Informationen von einem Ort an den anderen zu transportieren. Diese Informationen können gleichzeitig von verschiedenen Sensoren stammen. Sie müssen bearbeitet und in einen Netzwerkdatenstrom verwandelt werden. Gleichzeitig werden Daten von anderen Knoten empfangen und müssen bei Multihop-Routing an andere Knoten weitergereicht werden. Da die Speicherkapazität im Knoten selbst stark beschränkt ist, können Puffer nur sehr klein realisiert werden. Ein Betriebssystem ist in verschiedene Schichten unterteilt, zwischen denen es viele Datenflüsse gibt. Beim Empfang einer Nachricht zum Beispiel, müssen einige Events ausgelöst werden, bis die Nachricht auf der Anwendungsebene angekommen ist.

2.3 Vielfalt im Design und Anwendungsbereich

Sensornetzknoten sind von Natur aus Applikationsspezifisch. Es hat keinen Sinn ein Allzwecksystem zu entwickeln, wenn man auf die Grösse und den Energieverbrauch Wert legt. Man möchte ja auf jeglichen unnötigen Ballast verzichten. Da es unzählige Anwendungsgebiete gibt, ist auch die Anzahl der verschiedenen Hardwareausprägungen entsprechend gross. Deshalb ist auf Seiten des Betriebssystems ein erhöhtes Mass an Modularität gefragt, damit nur die benötigten Komponenten eingebaut, beziehungsweise installiert, werden müssen. Dies erfüllt am besten eine Entwicklungsumgebung, die es erlaubt, spezialisierte Anwendungen aus einem Spektrum von Modulen zu verwirklichen.

2.4 Verfügbarkeit

Die Sensorknoten sind zahlreich, unbeaufsichtigt und oft auch unzugänglich gelegen. Es ist deshalb nicht möglich, abgestürzte Knoten manuell zu „rebooten". Das Sensornetzwerk soll insgesamt eine hohe Verfügbarkeit haben, sonst ist es vielleicht gerade im entscheidenden Moment offline. Die Möglichkeit,

redundante Komponenten in den Knoten einzufügen, ist durch die engen Platz- und Energieverhältnisse beschränkt. Deshalb ist es vorteilhafter, die Redundanz knotenübergreifend zu realisieren. Durch gute Algorithmen kann man so das Ausfallen einzelner Knoten tolerieren. Es ist aber auch essentiell, die Zuverlässigkeit jedes einzelnen Knotens zu verbessern. Das Betriebssystem muss zuverlässig arbeiten, anders als wir uns das aus dem Alltag gewohnt sind. Zudem sollte das Betriebssystem die Entwicklung von zuverlässigen verteilten Applikationen ermöglichen und unterstützen.

3 TinyOS

Das Hauptproblem bei der Entwicklung eines Betriebssystems für Sensornetzwerke besteht darin, den vorgängig genannten Anforderungen unter den gezeigten Bedingungen nachzukommen. Die Hardwarekomponenten müssen effektiv verwaltet werden, um energieeffizient zu bleiben. In TinyOS wird ein eventbasiertes Modell eingesetzt, um einen hohen Parallelitätsgrad mit geringem Platzbedarf zu erreichen.

3.1 Der Aufbau von TinyOS

TinyOS ist folgendermassen aufgebaut: Im System gibt es einen Scheduler, der die verschiedenen Tasks verwaltet. Momentan ist ein einfacher FIFO-Scheduler implementiert. Das heisst, Tasks die lauffähig sind, werden in eine FIFO-Queue eingefügt. Bei einem Kontextwechsel wird einfach der vorderste Task der Queue aktiv gesetzt. Es soll aber ein prioritätsbasierter Scheduler entwickelt werden, um eine bessere Grundlage für Realtimeanwendungen zu bieten. Das restliche System ist in Komponenten aufgeteilt. Jede Komponente wird einer Schicht zugeordnet. Komponenten auf höheren Schichten rufen Commands auf, die von Komponenten auf tieferen Schichten zur Verfügung gestellt werden. Im Gegenzug benachrichtigen die Komponenten der unteren Schichten auftretende Ereignisse durch Events an die höher liegenden Komponenten. Die Hardwarekomponenten liegen in der untersten Schicht.

3.2 Komponenten

Jede Komponente besteht aus einem Frame, einigen Commands und mehreren Events, die ausgelöst werden können.

3.2.1 Frame

Jede Komponente erhält ein Speicherbereich fester Grösse. Somit ist der Speicherbedarf im Voraus bekannt. Es wird keine dynamische Allokation bereitgestellt.

3.2.2 Commands

Commands sind nicht-blockierende Anfragen an Komponenten in unteren Schichten. Ein Command legt die Anfrageparameter in seinem Frame ab und gründet einen Task für die spätere Ausführung. Ein Command muss seinem Caller eine Rückmeldung geben, ob die Aktion erfolgreich verlaufen ist.

3.2.3 Events

Event-Handler werden aufgerufen, um Hardwareevents bzw. Events von unteren Schichten abzuarbeiten. Ein Event-Handler kann Informationen in seinem Frame ablegen, Tasks gründen, höhere Ebenen mit Events benachrichtigen und Commands auf unteren Ebenen aufrufen.

3.2.4 Tasks

Tasks führen die eigentliche Arbeit aus. Sie sind atomar im Bezug auf andere Tasks. Sie können aber durch Events preempted werden. Tasks rufen Commands auf unteren Schichten auf, benachrichtigen höhere Schichten mit Events und schedulen andere Tasks in derselben Komponente. Weil Tasks immer zu Ende laufen, genügt es, in der Komponente einen einzigen Stack zu

haben, der jeweils dem laufenden Task zugeordnet ist. Dies ist in einem speicherbegrenzten System wichtig. Tasks erlauben es, verschiedene Komponenten quasi parallel laufen zu lassen, da sie durch Events asynchron miteinander kommunizieren. Das heisst eine Komponente muss nie blockierend auf eine andere Komponente warten, da sie jeweils mit Events benachrichtigt wird.

3.2.5 Einteilung der Komponenten

Es gibt drei Kategorien von Komponenten: Hardware-Abstraktionen, synthetische Hardware und Highlevel-Software-Komponenten.

Hardware-Abstraktionen bilden die Hardware in das Komponentenmodell ab. Zum Beispiel das RFM (Radio Frequency Module) gehört in diese Klasse. Diese Komponente bedient die Pins des RFM-Senders/Empfängers und informiert andere Komponenten durch Events, ob die Bits richtig gesendet wurden.

Synthetische Hardware simuliert das Verhalten von komplexer Hardware. Zum Beispiel die Radio Byte Komponente. Sie sendet Daten zum RFM und signalisiert, dass ein ganzes Byte gesendet wurde. Synthetische Hardware ist hilfreich, wenn man Experimente durchführt. Oft ist es nämlich nicht klar, ob man eine Komponente in Hard- oder Software haben will. Im Komponentenmodell können so Hardwarekomponenten durch Softwarekomponenten beliebig ausgetauscht werden.

Highlevel-Software-Komponenten kontrollieren andere Komponenten, berechnen das Routing und berechnen andere Datentransformationen. Ein Beispiel dafür ist das Messaging-Modul, das dafür verantwortlich ist, Nachrichten in Pakete zu zerstückeln und empfangene Pakete wieder zu Nachrichten zusammenzusetzen.

Das Komponentenmodell läst die Grenzen zwischen Hard- und Software verschwimmen. Durch die fixen Speicheranforderungen einer Komponente, ist es einfach, Hardware in das Modell abzubilden.

3.2.6 Eine Beispiel-Komponente

Eine typische Komponente ist die Messaging-Komponente. Sie exportiert Commands um die Komponente zu initialisieren und um die Power-Management Funktionen zu nutzen.

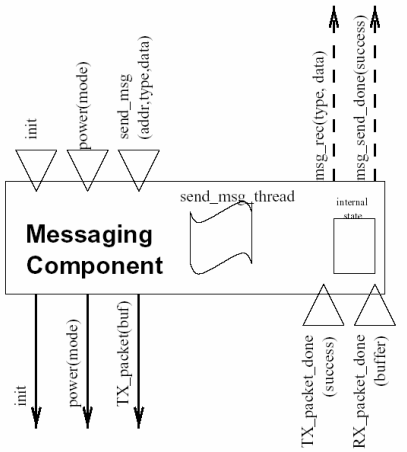

Abbildung 3:
Messaging-Komponente

```
//angebotene Commands
char TOS_COMMAND(send_msg)(int addr,int
type, char* data);
void TOS_COMMAND(powermode)(char mode);
char TOS_COMMAND(init)();

//ausgelöste Events:
char msg_rec(int type, char* data);
char msg_send_done(char success);

//behandelte Events:
char TX_packet_done(char success);
char RX_packet_done(char* packet);

//aufgerufene Commands:
char TOS_COMMAND(TX_packet)(char* data);
void TOS_COMMAND(powermode)(char mode);
char TOS_COMMAND(init)();
```

Abbildung 4:
Interface der Messaging-Komponente

Es gibt einen Command zu Beginn eines Sendevorgangs und Events, die das erfolgreiche Senden oder Empfangen einer Nachricht signalisieren. Während der Ausführung, ruft die Komponente Commands auf einer Komponente in der

Paketschicht auf und behandelt Events, die signalisieren, dass ein Paket gesendet oder empfangen wurde.

Durch die einfache Interfacebeschreibung ist das Zusammenbauen verschiedener Komponenten relative einfach. Die Kommunikation zwischen den Komponenten nimmt die Form von Funktionsaufrufen an und ist zur Compilezeit mit Typechecking einfach zu überprüfen

3.2.7 Zusammensetzen der Komponenten

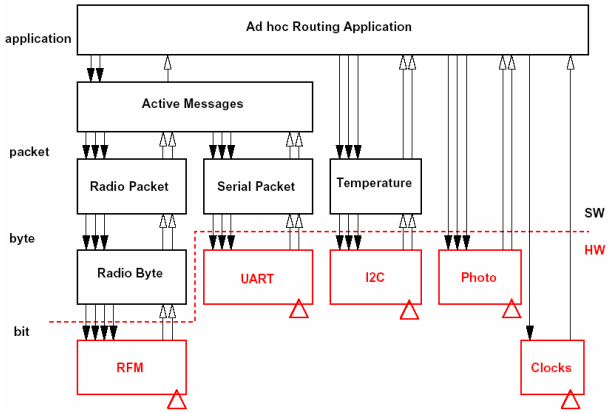

Abbildung 5:
Zusammenspiel der Komponenten

Nun wollen wir die verschiedenen Komponenten zusammensetzen. Hier am Beispiel eines Sensors, der periodisch die Lichtverhältnisse und die Temperatur misst und diese Daten danach weitersendet. Es gibt drei I/O-Schnittstellen: Den Temperatursensor, die Photodiode und die Netzwerkkomponente. Die oberste Schicht muss vier Events behandeln: Es gehen periodisch Routinginformationen ein, es gibt Pakete, die an andere weitergeleitet werden müssen und die Sensoren liefern Daten, die ständig verarbeitet werden müssen. Der Timerevent wird dazu benötigt, periodisch die Daten zu sammeln.

Wenn die Temperatur und das Licht gemessen wurden, wird der Sendmessage-Command der Messaging-Komponente aufgerufen. Danach wird ein Task gestartet, der mit dem Senden beauftragt ist. Es werden Pakete gebildet, die über die Paket-Komponente verschickt werden. Diese verwendet wiederum die Radio-Byte-Komponente, um die einzelnen Bytes des Paketes zu übermitteln. Zum Schluss versendet die Radio-Komponente die einzelnen Bits. Beim Empfangen von Bits wird die ganze Abfolge in umgekehrter Richtung durchlaufen. Wenn erst einmal die Adresse eines Paketes bekannt ist, so wird sie überprüft. Wenn das Paket für diesen Sensor bestimmt ist, so wird ein Event ausgelöst und der Dispatcher leitet das Paket zur richtigen Applikation weiter.

4 Virtuelle Maschinen in Sensornetzwerken

Da in Sensornetzen ganz unterschiedliche Hardware zum Einsatz kommt, die sich zudem stetig weiterentwickelt, ist es von Vorteil, eine einheitliche Hardware durch eine virtuelle Maschine zu beschreiben, auf der dann die verschiedenen Anwendungen laufen. Wenn man an Java denkt, so scheint ein zwei-mal-zwei Millimeter grosser Sensorknoten eine widrige Umgebung für eine virtuelle Maschine zu sein. Besonders wenn man an die beschränkten Ressourcen denkt. In der Tat ist es so, dass bestehende VMs in keiner Weise auf einen Sensorknoten passen.

Deshalb wird nun an der Universety of California Berkeley Maté entwickelt. Dies ist eine auf Sensorknoten getrimmte VM. Ein Sensorknoten hat 8 bis 128 kB Programmspeicher und 512 Byte bis 4 kB RAM. Die kleinste bestehende VM benötigt aber mindestens 160 kB. Ausserdem müssen wie überall die Energieaspekte berücksichtigt werden. Ein Bit zu senden kann gleich viel Energie benötigen wie 1000 Instruktionen auszuführen. Durch den Bytecodeinterpreter ist es auch einfach, mobilen Code zu realisieren. So ist es möglich, von aussen Programme auf dem Sensorknoten zu installieren. In vielen

Anwendungen von Sensornetzwerken ist es notwendig, die Knoten neu zu konfigurieren, da das weitere Vorgehen von den gemessenen Daten abhängt. Die Knoten können zum Reprogrammieren nicht einfach eingesammelt werden. Dazu muss ein effizienter Reprogrammiermechanismus zur Verfügung gestellt werden.

Der Bytecode ist im Allgemeinen viel kompakter als Maschinencode, da er viel abstrakter ist. Das heisst eine Bytecode-Instruktion kann bis zu 20'000 Maschineninstruktionen entsprechen. Es wurde gezeigt, dass die zusätzliche Energie, die gebraucht wird, um die VM am Laufen zu halten, durch die kleinere Nachrichtengrösse beim Übermitteln von Code kompensiert wird.

4.1 Maté

Maté ist ein Bytecode-Interpreter, der auf TinyOS läuft. Der Code wird in Capsules zu je 24 Instruktionen aufgeteilt. Maté passt in 1 kB RAM und 16 kB Programmspeicher.

4.1.1 Architektur

Das Kommunikationsmodell in Maté erlaubt es, eine Nachricht mit einem einzigen Befehl zu versenden. Die Nachricht wird dann automatisch geroutet. Die Maté-Instruktionen abstrahieren von der Asynchronität bei TinyOS. Wenn der Send-Befehl aufgerufen wird, so wird gewartet, bis das Acknowledgement angekommen ist. Wenn ein Sense-Befehl ausgeführt wird, so wird der Thread schlafen gelegt, bis die entsprechenden Daten anliegen. Diese Art zu programmieren ist viel weniger fehleranfällig, als wenn man das Ganze mit asynchronen Calls implementieren müsste. In Maté gibt es einen Operandenstack und einen Returnstack. Letzterer verwaltet die Rücksprungadressen. Es gibt eine gemeinsame Speicherzelle, die zur Kommunikation zwischen den verschiedenen Tasks dient.

4.1.2 Code Capsules

Maté Programme sind in Capsules von 24 Instruktionen aufgeteilt. Dies ist so gewählt, damit eine Capsule genau in ein TinyOS-Paket passt. Dies ist praktisch, da man beim Empfangen eines Programms die empfangenen Teil-Capsules nicht puffern muss, was Speicher spart.

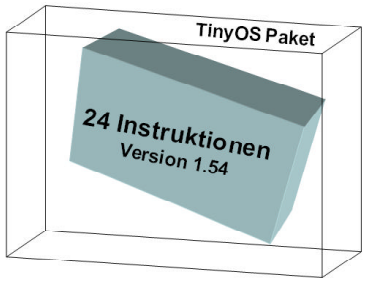

Abbildung 6: Tiny-OS Paket

Es gibt 4 Arten von Capsules:

- Messagesend-Capsules
- Messagereceive-Capsules
- Timer-Capsules
- Subroutine-Capsules

Subroutine-Capsules erlauben es, Programme zu haben, die grösser als 24 Instruktionen sind. Mit einer Call-Instruktion kann in den Code einer anderen Capsule gesprungen werden.

Capsules reagieren im Allgemeinen auf Events. Jede Capsule kann einem Event zugeordnet werden. Beim Auftreten eines Events wird die erste Instruktion der zugeordneten Capsule ausgeführt. Events werden unter anderem ausgelöst, wenn ein Timer abläuft, ein Paket eingeht oder ein Paket gesendet wird. Jede Bytecode-Instruktion wird als TinyOS-Task ausgeführt. Die Instruktionen sind ja komplexe Gebilde. Wenn eine Subroutine aufgerufen wird, so wird die aktuelle Adresse auf den Return-Stack gelegt und es wird zur Calls-Adresse gesprungen. Ist diese dann beendet, so wird die Rücksprungadresse vom Retrunstack gepopt und der PC wird auf diese Adresse zurückgesetzt.

Maté ist so ausgerichtet, dass fehlerhafte Capsules nicht zum Absturz des gesamten Systems führen. Es wird

einfach die fehlerhafte Capsule liquidiert und durch eine Version ersetzt, die nichts tut. Dies ist vor allem auch durch die hohe Abstraktion gegenüber den Lowlevel-Services, wie zum Beispiel der Netzschicht, möglich.

4.1.3 Viraler Code

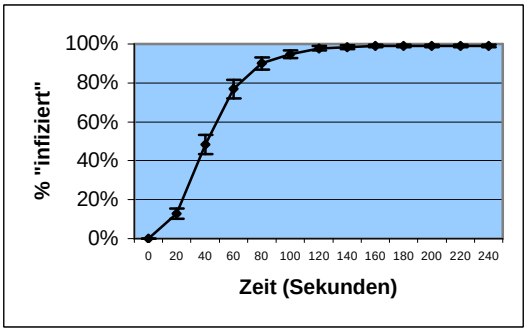

Abbildung 7:
Verbreitung von viralem Code

Da Code-Capsules genau in ein Tiny-OS-Paket passen, lassen sie sich einfach über das Netz verschicken. Dies kann dazu verwendet werden, ein neues Programm auf allen Knoten im Netz zu installieren – ohne manuelle Eingriffe. Um dies zu erreichen, wird ein Knoten mit der neuen Version „infiziert". Dieser sendet die neue Version an alle seine Nachbarn. Da jede Capsule eine Versionsnummer enthält, kann ein Knoten einfach feststellen, ob er bereits ein Update der entsprechenden Capsule erhalten hat. Ist dies nicht der Fall, so überschriebt er bei sich die alte Version und schickt die Capsule weiter. Falls er die Version schon hat, wird die Capsule vernichtet. Es ist so auch sichergestellt, dass kein Knoten eine alte Version bei sich installiert. Auf der abgebildeten Grafik sieht man den typischen Verlauf, wie sich eine neue Capsule auf dem Netz verbreitet. Am Anfang ist der Anstieg relativ steil. Mit der Zeit nimmt die Anzahl Neuinfektionen pro Zeit ab und die Kurve nähert sich asymptotisch 100%.

4.1.4 Energieaspekte

Man sieht, dass sich bei Maté ein Overhead gegenüber nativem Code ergibt.

Operation	Maté #CPU Zyklen	Native #CPU Zyklen	Verhältnis
Einfach `and`	469	14	**33.5:1**
OS Call `rand`	435	45	9.5:1
Sensoren `sense`	1342	396	3.4:1
Komplex `send`	685+~20'000	~20'000	**1.03:1**

Abbildung 8:
Verhältnis von Maté- zu CPU-Zyklen

Instruktionen, die Highlevel-Konstrukte anbieten sind effizient. Einfache Operationen dagegen haben einen grossen Overhead. Der Hauptgrund für diesen Overhead bei Maté ist, dass jeder Befehl in einem eigenen Task ausgeführt wird. Dies benötigt eine Queue- und Dequeu-Operation.

Der erhöhte Rechenaufwand von Maté führt zu einem Energie-Overhead. Dies wird aber wettgemacht, da die Programme viel kompakter verschickt werden können. Der Bytecode ist etwa 100-mal kleiner als der binäre Maschinencode. Es ist klar, dass es hier darauf ankommt, wie oft eine Capsule ausgeführt wird. Werden immer neue Capsules herumgeschickt, so ist Maté billiger. Ist dies nicht der Fall, so wählt man besser den Nativecode. Auf der Great Duck Insel hat man festgestgestellt, dass es erst effizienter ist Nativecode zu verwenden, wenn eine Applikation länger als 5 Tage läuft.

4.1.5 Schutzmechanismen

Da der TinyOS-Kernel nicht die Schutzmechanismen eines konventionellen Betriebssystems aufweist, ist es sehr wahrscheinlich, dass fehlerhafte Programme den ganzen Knoten zum Absturz bringen. Dies wird durch Maté behoben, da es Highlevel-Konstrukte zur Verfügung stellt und keine direkten Zugriffe auf Interruptus und Speicher-

zellen zulässt. In einem Knoten ist kaum Platz für hardwaremässig unterstütztes Virtual-Memory. Da Maté eine virtuelle Maschine ist, kann so etwas realisiert werden. Ausserdem ist eine abstrakte Sprache viel einfacher zu bedienen und führt zu einer weiteren Verbreitung von Sensornetzwerken, da auch Nicht-Spezialisten einfache Programme schreiben können.

4.1.6 Ein einfaches Maté-Programm

Das Programm liest periodisch den Lichtsensor aus. Wenn der gelesene Wert mehr als 32 vom zuletzt versendeten Wert abweicht, so wird der neue Wert gesendet. Dieses Programm passt in eine einzige Capsule.

```
00  pushc 1     # 1 auf Operandenstack pushen
01  sense       # Sensor 1 auslesen (Licht)
02  copy        # Sensorwert auf Stack
03  gets        # zuletzt gesendeter Wert auf Stack
04  inv         # letzten Wert invertieren
05  add         # dif = Sensorwert + (–letzter Wert)
06  pushc 32
07  add         # dif = dif + 32
08  blez 17     # Sensorwert zu klein: springe zu send
09  copy        # Sensorwert auslesen
10  inv         # Sensorwert Wert invertieren
11  gets        # zuletzt gesendeter Wert auf Stack
12  add         # dif = -Sensorwert + letzter Wert
13  pushc 32
14  add         # dif = dif + 32
15  blez 17     # Sensorwert zu gross: springe zu send
16  halt
17  copy        # Sensorwert auslesen
18  sets        # neuen Wert abspeichern
19  pushm       # Nachricht auf Operandenstack
20  clear       # Nachrichteninhalt löschen
21  add         # Nachrichteninhalt reinkopieren
22  send        # Nachricht versenden
23  halt
```

5 Quellen

[1] Philip Levis and David Culler: Maté: A Tiny Virtual Machine for Sensor Networks

[2] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister: System Architecture Directions for Networked Sensors

[3] Phillip Stanley-Marbell and Liviu Iftode: Scylla: A Smart Virtual Machine for Mobile Embedded Systems

[4] TinyOS Webseite: http://webs.cs.berkeley.edu/tos